Table of Contents

TensorFlow Examples

> From: aymericdamien/TensorFlow-Examples

This tutorial was designed for easily diving into TensorFlow, through examples. For readability, it includes both notebooks and source codes with explanation.

It is suitable for beginners who want to find clear and concise examples about TensorFlow. Besides the traditional 'raw' TensorFlow implementations, you can also find the latest TensorFlow API practices (such as `layers` , `estimator` , `dataset` , ...).

Update (03/18/2018): TensorFlow's Eager API examples available! (TF v1.5+ recommended).

If you are using older TensorFlow version (0.11 and under), please have a look here.

Prerequisite

Machine Learning

Prior to start browsing the examples, it may be useful that you get familiar with machine learning, as TensorFlow is mostly used for machine learning tasks (especially Neural Networks). You can find below a list of useful links, that can give you the basic knowledge required for this TensorFlow Tutorial.

Machine Learning

- An Introduction to Machine Learning Theory and Its Applications: A Visual Tutorial with Examples
- A Gentle Guide to Machine Learning
- A Visual Introduction to Machine Learning
- Introduction to Machine Learning

Deep Learning & Neural Networks

- An Introduction to Neural Networks
- An Introduction to Image Recognition with Deep Learning
- Neural Networks and Deep Learning

MNIST Dataset Introduction

Most examples are using MNIST dataset of handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flatten and converted to a 1-D numpy array of 784 features (28*28).

Overview

Usage

In our examples, we are using TensorFlow input_data.py script to load that dataset. It is quite useful for managing our data, and handle:

- Dataset downloading
- Loading the entire dataset into numpy array:

```
# Import MNIST
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Load data
X_train = mnist.train.images
Y_train = mnist.train.labels
X_test = mnist.test.images
Y_test = mnist.test.labels
```

- A `next_batch` function that can iterate over the whole dataset and return only the desired fraction of the dataset samples (in order to save memory and avoid to load the entire dataset).

```
# Get the next 64 images array and labels
batch_X, batch_Y = mnist.train.next_batch(64)
```

Link: http://yann.lecun.com/exdb/mnist/

Introduction

```python
import tensorflow as tf
```

```python
# Simple hello world using TensorFlow

# Create a Constant op
# The op is added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.

hello = tf.constant('Hello, TensorFlow!')
```

```python
# Start tf session
sess = tf.Session()
```

```python
# Run graph
print sess.run(hello)
```

```
Hello, TensorFlow!
```

```
# Basic Operations example using TensorFlow library.
# Author: Aymeric Damien
# Project: https://github.com/aymericdamien/TensorFlow-Examples/
```

```
import tensorflow as tf
```

```
# Basic constant operations
# The value returned by the constructor represents the output
# of the Constant op.
a = tf.constant(2)
b = tf.constant(3)
```

```
# Launch the default graph.
with tf.Session() as sess:
    print "a: %i" % sess.run(a), "b: %i" % sess.run(b)
    print "Addition with constants: %i" % sess.run(a+b)
    print "Multiplication with constants: %i" % sess.run(a*b)
```

```
a=2, b=3
Addition with constants: 5
Multiplication with constants: 6
```

```
# Basic Operations with variable as graph input
# The value returned by the constructor represents the output
# of the Variable op. (define as input when running session)
# tf Graph input
a = tf.placeholder(tf.int16)
b = tf.placeholder(tf.int16)
```

```
# Define some operations
add = tf.add(a, b)
mul = tf.multiply(a, b)
```

```
# Launch the default graph.
with tf.Session() as sess:
    # Run every operation with variable input
    print "Addition with variables: %i" % sess.run(add, feed_dic
t={a: 2, b: 3})
    print "Multiplication with variables: %i" % sess.run(mul, fe
ed_dict={a: 2, b: 3})
```

```
Addition with variables: 5
Multiplication with variables: 6
```

```
# ----------------
# More in details:
# Matrix Multiplication from TensorFlow official tutorial

# Create a Constant op that produces a 1x2 matrix.  The op is
# added as a node to the default graph.
#
# The value returned by the constructor represents the output
# of the Constant op.
matrix1 = tf.constant([[3., 3.]])
```

```
# Create another Constant that produces a 2x1 matrix.
matrix2 = tf.constant([[2.],[2.]])
```

```
# Create a Matmul op that takes 'matrix1' and 'matrix2' as input
s.
# The returned value, 'product', represents the result of the ma
trix
# multiplication.
product = tf.matmul(matrix1, matrix2)
```

```
# To run the matmul op we call the session 'run()' method, passi
ng 'product'
# which represents the output of the matmul op.  This indicates
to the call
# that we want to get the output of the matmul op back.
#
# All inputs needed by the op are run automatically by the sessi
on.  They
# typically are run in parallel.
#
# The call 'run(product)' thus causes the execution of threes op
s in the
# graph: the two constants and matmul.
#
# The output of the op is returned in 'result' as a numpy `ndarr
ay` object.
with tf.Session() as sess:
    result = sess.run(product)
    print result
```

```
[[ 12.]]
```

Basic introduction to TensorFlow's Eager API

A simple introduction to get started with TensorFlow's Eager API.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

What is TensorFlow's Eager API ?

Eager execution is an imperative, define-by-run interface where operations are executed immediately as they are called from Python. This makes it easier to get started with TensorFlow, and can make research and development more intuitive. A vast majority of the TensorFlow API remains the same whether eager execution is enabled or not. As a result, the exact same code that constructs TensorFlow graphs (e.g. using the layers API) can be executed imperatively by using eager execution. Conversely, most models written with Eager enabled can be converted to a graph that can be further optimized and/or extracted for deployment in production without changing code. - Rajat Monga

More info: https://research.googleblog.com/2017/10/eager-execution-imperative-define-by.html

```
from __future__ import absolute_import, division, print_function

import numpy as np
import tensorflow as tf
import tensorflow.contrib.eager as tfe
```

```
# Set Eager API
print("Setting Eager mode...")
tfe.enable_eager_execution()
```

```
Setting Eager mode...
```

```
# Define constant tensors
print("Define constant tensors")
a = tf.constant(2)
print("a = %i" % a)
b = tf.constant(3)
print("b = %i" % b)
```

```
Define constant tensors
a = 2
b = 3
```

```
# Run the operation without the need for tf.Session
print("Running operations, without tf.Session")
c = a + b
print("a + b = %i" % c)
d = a * b
print("a * b = %i" % d)
```

```
Running operations, without tf.Session
a + b = 5
a * b = 6
```

```
# Full compatibility with Numpy
print("Mixing operations with Tensors and Numpy Arrays")

# Define constant tensors
a = tf.constant([[2., 1.],
                 [1., 0.]], dtype=tf.float32)
print("Tensor:\n a = %s" % a)
b = np.array([[3., 0.],
              [5., 1.]], dtype=np.float32)
print("NumpyArray:\n b = %s" % b)
```

```
Mixing operations with Tensors and Numpy Arrays
Tensor:
 a = tf.Tensor(
[[2. 1.]
 [1. 0.]], shape=(2, 2), dtype=float32)
NumpyArray:
 b = [[3. 0.]
 [5. 1.]]
```

```
# Run the operation without the need for tf.Session
print("Running operations, without tf.Session")

c = a + b
print("a + b = %s" % c)

d = tf.matmul(a, b)
print("a * b = %s" % d)
```

```
Running operations, without tf.Session
a + b = tf.Tensor(
[[5. 1.]
 [6. 1.]], shape=(2, 2), dtype=float32)
a * b = tf.Tensor(
[[11.  1.]
 [ 3.  0.]], shape=(2, 2), dtype=float32)
```

```
print("Iterate through Tensor 'a':")
for i in range(a.shape[0]):
    for j in range(a.shape[1]):
        print(a[i][j])
```

```
Iterate through Tensor 'a':
tf.Tensor(2.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
tf.Tensor(0.0, shape=(), dtype=float32)
```

Basic Models

Linear Regression Example

A linear regression learning algorithm example using TensorFlow library.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
import tensorflow as tf
import numpy
import matplotlib.pyplot as plt
rng = numpy.random
```

```
# Parameters
learning_rate = 0.01
training_epochs = 1000
display_step = 50
```

```
# Training Data
train_X = numpy.asarray([3.3,4.4,5.5,6.71,6.93,4.168,9.779,6.182,
7.59,2.167,
                         7.042,10.791,5.313,7.997,5.654,9.27,3.1
])
train_Y = numpy.asarray([1.7,2.76,2.09,3.19,1.694,1.573,3.366,2.
596,2.53,1.221,
                         2.827,3.465,1.65,2.904,2.42,2.94,1.3])
n_samples = train_X.shape[0]
```

```
# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="weight")
b = tf.Variable(rng.randn(), name="bias")
```

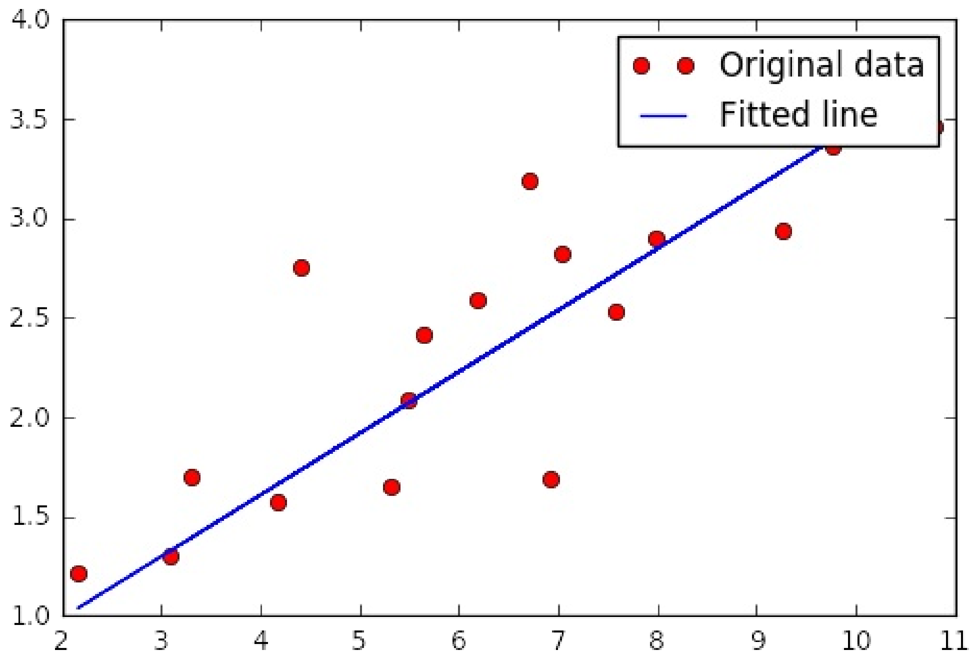
```
# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)
```

```
# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
# Gradient descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).min
imize(cost)
```

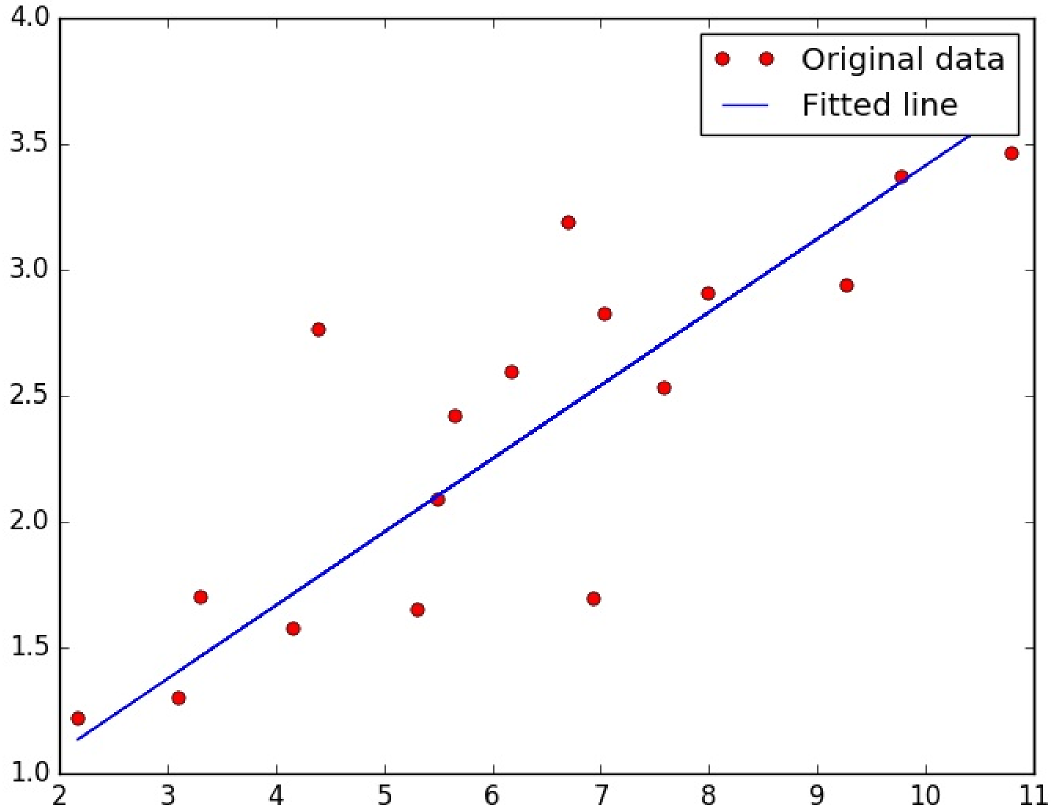
```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        #Display logs per epoch step
        if (epoch+1) % display_step == 0:
            c = sess.run(cost, feed_dict={X: train_X, Y:train_Y}
)
            print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}"
.format(c), \
                "W=", sess.run(W), "b=", sess.run(b)

    print "Optimization Finished!"
    training_cost = sess.run(cost, feed_dict={X: train_X, Y: tra
in_Y})
    print "Training cost=", training_cost, "W=", sess.run(W), "b
=", sess.run(b), '\n'

    #Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label=
'Fitted line')
    plt.legend()
    plt.show()
```

```
Epoch: 0050 cost= 0.195095107 W= 0.441748 b= -0.580876
Epoch: 0100 cost= 0.181448311 W= 0.430319 b= -0.498661
Epoch: 0150 cost= 0.169377610 W= 0.419571 b= -0.421336
Epoch: 0200 cost= 0.158700854 W= 0.409461 b= -0.348611
Epoch: 0250 cost= 0.149257123 W= 0.399953 b= -0.28021
Epoch: 0300 cost= 0.140904188 W= 0.391011 b= -0.215878
Epoch: 0350 cost= 0.133515999 W= 0.3826 b= -0.155372
Epoch: 0400 cost= 0.126981199 W= 0.374689 b= -0.0984639
Epoch: 0450 cost= 0.121201262 W= 0.367249 b= -0.0449408
Epoch: 0500 cost= 0.116088994 W= 0.360252 b= 0.00539905
Epoch: 0550 cost= 0.111567356 W= 0.35367 b= 0.052745
Epoch: 0600 cost= 0.107568085 W= 0.34748 b= 0.0972751
Epoch: 0650 cost= 0.104030922 W= 0.341659 b= 0.139157
Epoch: 0700 cost= 0.100902475 W= 0.336183 b= 0.178547
Epoch: 0750 cost= 0.098135538 W= 0.331033 b= 0.215595
Epoch: 0800 cost= 0.095688373 W= 0.32619 b= 0.25044
Epoch: 0850 cost= 0.093524046 W= 0.321634 b= 0.283212
Epoch: 0900 cost= 0.091609895 W= 0.317349 b= 0.314035
Epoch: 0950 cost= 0.089917004 W= 0.31332 b= 0.343025
Epoch: 1000 cost= 0.088419855 W= 0.30953 b= 0.370291
Optimization Finished!
Training cost= 0.0884199 W= 0.30953 b= 0.370291
```




```
# Regression result
```



Original data
Fitted line
4.0
3.5
3.0
2.5
2.0
1.5
1.0
2
3
4
5
6
7
8
9
10
11

Linear Regression with Eager API

A linear regression implemented using TensorFlow's Eager API.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import absolute_import, division, print_function

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow.contrib.eager as tfe
```

```
# Set Eager API
tfe.enable_eager_execution()
```

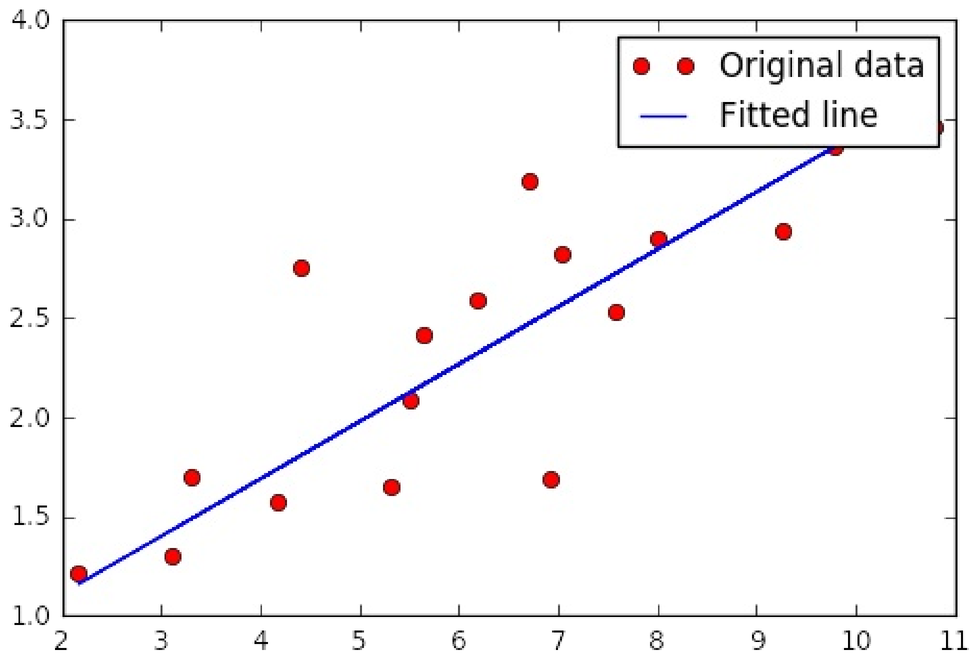
```
# Training Data
train_X = [3.3, 4.4, 5.5, 6.71, 6.93, 4.168, 9.779, 6.182, 7.59, 2.167,
           7.042, 10.791, 5.313, 7.997, 5.654, 9.27, 3.1]
train_Y = [1.7, 2.76, 2.09, 3.19, 1.694, 1.573, 3.366, 2.596, 2.53, 1.221,
           2.827, 3.465, 1.65, 2.904, 2.42, 2.94, 1.3]
n_samples = len(train_X)

# Parameters
learning_rate = 0.01
display_step = 100
num_steps = 1000
```

```
# Weight and Bias
W = tfe.Variable(np.random.randn())
b = tfe.Variable(np.random.randn())

# Linear regression (Wx + b)
def linear_regression(inputs):
    return inputs * W + b

# Mean square error
def mean_square_fn(model_fn, inputs, labels):
    return tf.reduce_sum(tf.pow(model_fn(inputs) - labels, 2)) /
 (2 * n_samples)
```

```
# SGD Optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lear
ning_rate)

# Compute gradients
grad = tfe.implicit_gradients(mean_square_fn)
```

```
# Initial cost, before optimizing
print("Initial cost= {:.9f}".format(
    mean_square_fn(linear_regression, train_X, train_Y)),
    "W=", W.numpy(), "b=", b.numpy())

# Training
for step in range(num_steps):

    optimizer.apply_gradients(grad(linear_regression, train_X, t
rain_Y))

    if (step + 1) % display_step == 0 or step == 0:
        print("Epoch:", '%04d' % (step + 1), "cost=",
              "{:.9f}".format(mean_square_fn(linear_regression,
train_X, train_Y)),
              "W=", W.numpy(), "b=", b.numpy())

# Graphic display
plt.plot(train_X, train_Y, 'ro', label='Original data')
plt.plot(train_X, np.array(W * train_X + b), label='Fitted line'
)
plt.legend()
plt.show()
```

```
Initial cost= 31.307329178 W= -0.7870768 b= -0.2507985
Epoch: 0001 cost= 9.502781868 W= -0.26173288 b= -0.17560114
Epoch: 0100 cost= 0.114994615 W= 0.36224815 b= 0.014603348
Epoch: 0200 cost= 0.106785327 W= 0.34959725 b= 0.104292504
Epoch: 0300 cost= 0.100346453 W= 0.33839324 b= 0.1837239
Epoch: 0400 cost= 0.095296182 W= 0.32847065 b= 0.25407064
Epoch: 0500 cost= 0.091335081 W= 0.3196829 b= 0.3163719
Epoch: 0600 cost= 0.088228233 W= 0.31190023 b= 0.37154746
Epoch: 0700 cost= 0.085791394 W= 0.30500764 b= 0.42041263
Epoch: 0800 cost= 0.083880097 W= 0.2989034 b= 0.46368918
Epoch: 0900 cost= 0.082380980 W= 0.2934973 b= 0.50201607
Epoch: 1000 cost= 0.081205189 W= 0.28870946 b= 0.5359594
```



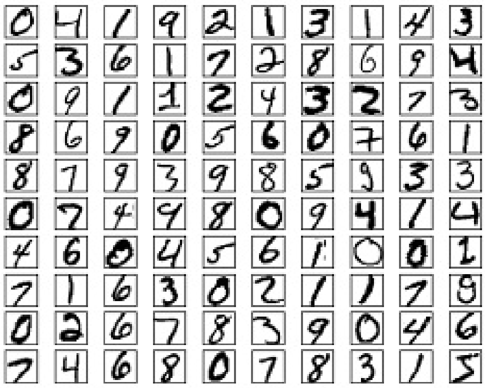
Logistic Regression Example

A logistic regression learning algorithm example using TensorFlow library.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).



More info: http://yann.lecun.com/exdb/mnist/

```
import tensorflow as tf

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder(tf.float32, [None, 784]) # mnist data image o
f shape 28*28=784
y = tf.placeholder(tf.float32, [None, 10]) # 0-9 digits recognit
ion => 10 classes

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

# Minimize error using cross entropy
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_i
ndices=1))
# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).min
imize(cost)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_si
ze)
            # Fit training using batch data
            _, c = sess.run([optimizer, cost], feed_dict={x: bat
ch_xs,
                                                          y: bat
ch_ys})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}"
.format(avg_cost)

    print "Optimization Finished!"

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(
y, 1))
    # Calculate accuracy for 3000 examples
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.flo
at32))
    print "Accuracy:", accuracy.eval({x: mnist.test.images[:3000
], y: mnist.test.labels[:3000]})
```

```
Epoch: 0001 cost= 1.182138959
Epoch: 0002 cost= 0.664778162
Epoch: 0003 cost= 0.552686284
Epoch: 0004 cost= 0.498628905
Epoch: 0005 cost= 0.465469866
Epoch: 0006 cost= 0.442537872
Epoch: 0007 cost= 0.425462044
Epoch: 0008 cost= 0.412185303
Epoch: 0009 cost= 0.401311587
Epoch: 0010 cost= 0.392326203
Epoch: 0011 cost= 0.384736038
Epoch: 0012 cost= 0.378137191
Epoch: 0013 cost= 0.372363752
Epoch: 0014 cost= 0.367308579
Epoch: 0015 cost= 0.362704660
Epoch: 0016 cost= 0.358588599
Epoch: 0017 cost= 0.354823110
```

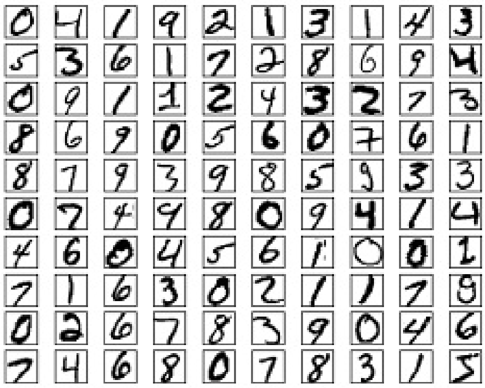
Logistic Regression with Eager API

A logistic regression implemented using TensorFlow's Eager API.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).



More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import absolute_import, division, print_function

import tensorflow as tf
import tensorflow.contrib.eager as tfe
```

```
# Set Eager API
tfe.enable_eager_execution()
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.1
batch_size = 128
num_steps = 1000
display_step = 100
```

```
# Iterator for the dataset
dataset = tf.data.Dataset.from_tensor_slices(
    (mnist.train.images, mnist.train.labels)).batch(batch_size)
dataset_iter = tfe.Iterator(dataset)
```

```
# Variables
W = tfe.Variable(tf.zeros([784, 10]), name='weights')
b = tfe.Variable(tf.zeros([10]), name='bias')

# Logistic regression (Wx + b)
def logistic_regression(inputs):
    return tf.matmul(inputs, W) + b

# Cross-Entropy loss function
def loss_fn(inference_fn, inputs, labels):
    # Using sparse_softmax cross entropy
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_wit
h_logits(
        logits=inference_fn(inputs), labels=labels))

# Calculate accuracy
def accuracy_fn(inference_fn, inputs, labels):
    prediction = tf.nn.softmax(inference_fn(inputs))
    correct_pred = tf.equal(tf.argmax(prediction, 1), labels)
    return tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

```
# SGD Optimizer
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lear
ning_rate)

# Compute gradients
grad = tfe.implicit_gradients(loss_fn)
```

```
# Training
average_loss = 0.
average_acc = 0.
for step in range(num_steps):

    # Iterate through the dataset
    try:
        d = dataset_iter.next()
    except StopIteration:
        # Refill queue
        dataset_iter = tfe.Iterator(dataset)
        d = dataset_iter.next()

    # Images
    x_batch = d[0]
    # Labels
    y_batch = tf.cast(d[1], dtype=tf.int64)

    # Compute the batch loss
    batch_loss = loss_fn(logistic_regression, x_batch, y_batch)
    average_loss += batch_loss
    # Compute the batch accuracy
    batch_accuracy = accuracy_fn(logistic_regression, x_batch, y
_batch)
    average_acc += batch_accuracy

    if step == 0:
        # Display the initial cost, before optimizing
        print("Initial loss= {:.9f}".format(average_loss))

    # Update the variables following gradients info
    optimizer.apply_gradients(grad(logistic_regression, x_batch,
 y_batch))

    # Display info
    if (step + 1) % display_step == 0 or step == 0:
        if step > 0:
            average_loss /= display_step
            average_acc /= display_step
        print("Step:", '%04d' % (step + 1), " loss=",
              "{:.9f}".format(average_loss), " accuracy=",
              "{:.4f}".format(average_acc))
        average_loss = 0.
        average_acc = 0.
```

```
Initial loss= 2.302584887
Step: 0001  loss= 2.302584887  accuracy= 0.1172
Step: 0100  loss= 0.952338457  accuracy= 0.7955
Step: 0200  loss= 0.535867393  accuracy= 0.8712
Step: 0300  loss= 0.485415280  accuracy= 0.8757
Step: 0400  loss= 0.433947206  accuracy= 0.8843
Step: 0500  loss= 0.381990731  accuracy= 0.8971
Step: 0600  loss= 0.394154936  accuracy= 0.8947
Step: 0700  loss= 0.391497582  accuracy= 0.8905
Step: 0800  loss= 0.386373103  accuracy= 0.8945
Step: 0900  loss= 0.332039326  accuracy= 0.9096
Step: 1000  loss= 0.358993769  accuracy= 0.9002
```

```
# Evaluate model on the test image set
testX = mnist.test.images
testY = mnist.test.labels

test_acc = accuracy_fn(logistic_regression, testX, testY)
print("Testset Accuracy: {:.4f}".format(test_acc))
```

```
Testset Accuracy: 0.9083
```

Nearest Neighbor Example

A nearest neighbor learning algorithm example using TensorFlow library. This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/)

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
import numpy as np
import tensorflow as tf

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
# In this example, we limit mnist data
Xtr, Ytr = mnist.train.next_batch(5000) #5000 for training (nn c
andidates)
Xte, Yte = mnist.test.next_batch(200) #200 for testing

# tf Graph Input
xtr = tf.placeholder("float", [None, 784])
xte = tf.placeholder("float", [784])

# Nearest Neighbor calculation using L1 Distance
# Calculate L1 Distance
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte))),
reduction_indices=1)
# Prediction: Get min distance index (Nearest neighbor)
pred = tf.arg_min(distance, 0)

accuracy = 0.

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:
    sess.run(init)

    # loop over test data
    for i in range(len(Xte)):
        # Get nearest neighbor
        nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: Xte[
i, :]})
        # Get nearest neighbor class label and compare it to its
 true label
        print "Test", i, "Prediction:", np.argmax(Ytr[nn_index])
, \
            "True Class:", np.argmax(Yte[i])
        # Calculate accuracy
        if np.argmax(Ytr[nn_index]) == np.argmax(Yte[i]):
            accuracy += 1./len(Xte)
    print "Done!"
    print "Accuracy:", accuracy
```

```
Test 0 Prediction: 7 True Class: 7
Test 1 Prediction: 2 True Class: 2
Test 2 Prediction: 1 True Class: 1
Test 3 Prediction: 0 True Class: 0
Test 4 Prediction: 4 True Class: 4
Test 5 Prediction: 1 True Class: 1
Test 6 Prediction: 4 True Class: 4
Test 7 Prediction: 9 True Class: 9
Test 8 Prediction: 8 True Class: 5
Test 9 Prediction: 9 True Class: 9
Test 10 Prediction: 0 True Class: 0
Test 11 Prediction: 0 True Class: 6
Test 12 Prediction: 9 True Class: 9
Test 13 Prediction: 0 True Class: 0
Test 14 Prediction: 1 True Class: 1
Test 15 Prediction: 5 True Class: 5
Test 16 Prediction: 4 True Class: 9
Test 17 Prediction: 7 True Class: 7
Test 18 Prediction: 3 True Class: 3
Test 19 Prediction: 4 True Class: 4
Test 20 Prediction: 9 True Class: 9
Test 21 Prediction: 6 True Class: 6
Test 22 Prediction: 6 True Class: 6
Test 23 Prediction: 5 True Class: 5
Test 24 Prediction: 4 True Class: 4
Test 25 Prediction: 0 True Class: 0
Test 26 Prediction: 7 True Class: 7
Test 27 Prediction: 4 True Class: 4
Test 28 Prediction: 0 True Class: 0
Test 29 Prediction: 1 True Class: 1
```

```
Test 30 Prediction: 3 True Class: 3
Test 31 Prediction: 1 True Class: 1
Test 32 Prediction: 3 True Class: 3
Test 33 Prediction: 4 True Class: 4
Test 34 Prediction: 7 True Class: 7
Test 35 Prediction: 2 True Class: 2
Test 36 Prediction: 7 True Class: 7
Test 37 Prediction: 1 True Class: 1
Test 38 Prediction: 2 True Class: 2
Test 39 Prediction: 1 True Class: 1
Test 40 Prediction: 1 True Class: 1
Test 41 Prediction: 7 True Class: 7
Test 42 Prediction: 4 True Class: 4
Test 43 Prediction: 1 True Class: 2
Test 44 Prediction: 3 True Class: 3
Test 45 Prediction: 5 True Class: 5
Test 46 Prediction: 1 True Class: 1
Test 47 Prediction: 2 True Class: 2
Test 48 Prediction: 4 True Class: 4
Test 49 Prediction: 4 True Class: 4
Test 50 Prediction: 6 True Class: 6
Test 51 Prediction: 3 True Class: 3
Test 52 Prediction: 5 True Class: 5
Test 53 Prediction: 5 True Class: 5
Test 54 Prediction: 6 True Class: 6
Test 55 Prediction: 0 True Class: 0
Test 56 Prediction: 4 True Class: 4
Test 57 Prediction: 1 True Class: 1
Test 58 Prediction: 9 True Class: 9
Test 59 Prediction: 5 True Class: 5
Test 60 Prediction: 7 True Class: 7
Test 61 Prediction: 8 True Class: 8
Test 62 Prediction: 9 True Class: 9
Test 63 Prediction: 3 True Class: 3
Test 64 Prediction: 7 True Class: 7
Test 65 Prediction: 4 True Class: 4
Test 66 Prediction: 6 True Class: 6
Test 67 Prediction: 4 True Class: 4
Test 68 Prediction: 3 True Class: 3
Test 69 Prediction: 0 True Class: 0
Test 70 Prediction: 7 True Class: 7
Test 71 Prediction: 0 True Class: 0
Test 72 Prediction: 2 True Class: 2
Test 73 Prediction: 7 True Class: 9
Test 74 Prediction: 1 True Class: 1
Test 75 Prediction: 7 True Class: 7
Test 76 Prediction: 3 True Class: 3
Test 77 Prediction: 7 True Class: 2
Test 78 Prediction: 9 True Class: 9
Test 79 Prediction: 7 True Class: 7
Test 80 Prediction: 7 True Class: 7
Test 81 Prediction: 6 True Class: 6
Test 82 Prediction: 2 True Class: 2
```

```
Test 83 Prediction: 7 True Class: 7
Test 84 Prediction: 8 True Class: 8
Test 85 Prediction: 4 True Class: 4
Test 86 Prediction: 7 True Class: 7
Test 87 Prediction: 3 True Class: 3
Test 88 Prediction: 6 True Class: 6
Test 89 Prediction: 1 True Class: 1
Test 90 Prediction: 3 True Class: 3
Test 91 Prediction: 6 True Class: 6
Test 92 Prediction: 9 True Class: 9
Test 93 Prediction: 3 True Class: 3
Test 94 Prediction: 1 True Class: 1
Test 95 Prediction: 4 True Class: 4
Test 96 Prediction: 1 True Class: 1
Test 97 Prediction: 7 True Class: 7
Test 98 Prediction: 6 True Class: 6
Test 99 Prediction: 9 True Class: 9
Test 100 Prediction: 6 True Class: 6
Test 101 Prediction: 0 True Class: 0
Test 102 Prediction: 5 True Class: 5
Test 103 Prediction: 4 True Class: 4
Test 104 Prediction: 9 True Class: 9
Test 105 Prediction: 9 True Class: 9
Test 106 Prediction: 2 True Class: 2
Test 107 Prediction: 1 True Class: 1
Test 108 Prediction: 9 True Class: 9
Test 109 Prediction: 4 True Class: 4
Test 110 Prediction: 8 True Class: 8
Test 111 Prediction: 7 True Class: 7
Test 112 Prediction: 3 True Class: 3
Test 113 Prediction: 9 True Class: 9
Test 114 Prediction: 7 True Class: 7
Test 115 Prediction: 9 True Class: 4
Test 116 Prediction: 9 True Class: 4
Test 117 Prediction: 4 True Class: 4
Test 118 Prediction: 9 True Class: 9
Test 119 Prediction: 7 True Class: 2
Test 120 Prediction: 5 True Class: 5
Test 121 Prediction: 4 True Class: 4
Test 122 Prediction: 7 True Class: 7
Test 123 Prediction: 6 True Class: 6
Test 124 Prediction: 7 True Class: 7
Test 125 Prediction: 9 True Class: 9
Test 126 Prediction: 0 True Class: 0
Test 127 Prediction: 5 True Class: 5
Test 128 Prediction: 8 True Class: 8
Test 129 Prediction: 5 True Class: 5
Test 130 Prediction: 6 True Class: 6
Test 131 Prediction: 6 True Class: 6
Test 132 Prediction: 5 True Class: 5
Test 133 Prediction: 7 True Class: 7
Test 134 Prediction: 8 True Class: 8
Test 135 Prediction: 1 True Class: 1
```

```
Test 136 Prediction: 0 True Class: 0
Test 137 Prediction: 1 True Class: 1
Test 138 Prediction: 6 True Class: 6
Test 139 Prediction: 4 True Class: 4
Test 140 Prediction: 6 True Class: 6
Test 141 Prediction: 7 True Class: 7
Test 142 Prediction: 2 True Class: 3
Test 143 Prediction: 1 True Class: 1
Test 144 Prediction: 7 True Class: 7
Test 145 Prediction: 1 True Class: 1
Test 146 Prediction: 8 True Class: 8
Test 147 Prediction: 2 True Class: 2
Test 148 Prediction: 0 True Class: 0
Test 149 Prediction: 1 True Class: 2
Test 150 Prediction: 9 True Class: 9
Test 151 Prediction: 9 True Class: 9
Test 152 Prediction: 5 True Class: 5
Test 153 Prediction: 5 True Class: 5
Test 154 Prediction: 1 True Class: 1
Test 155 Prediction: 5 True Class: 5
Test 156 Prediction: 6 True Class: 6
Test 157 Prediction: 0 True Class: 0
Test 158 Prediction: 3 True Class: 3
Test 159 Prediction: 4 True Class: 4
Test 160 Prediction: 4 True Class: 4
Test 161 Prediction: 6 True Class: 6
Test 162 Prediction: 5 True Class: 5
Test 163 Prediction: 4 True Class: 4
Test 164 Prediction: 6 True Class: 6
Test 165 Prediction: 5 True Class: 5
Test 166 Prediction: 4 True Class: 4
Test 167 Prediction: 5 True Class: 5
Test 168 Prediction: 1 True Class: 1
Test 169 Prediction: 4 True Class: 4
Test 170 Prediction: 9 True Class: 4
Test 171 Prediction: 7 True Class: 7
Test 172 Prediction: 2 True Class: 2
Test 173 Prediction: 3 True Class: 3
Test 174 Prediction: 2 True Class: 2
Test 175 Prediction: 1 True Class: 7
Test 176 Prediction: 1 True Class: 1
Test 177 Prediction: 8 True Class: 8
Test 178 Prediction: 1 True Class: 1
Test 179 Prediction: 8 True Class: 8
Test 180 Prediction: 1 True Class: 1
Test 181 Prediction: 8 True Class: 8
Test 182 Prediction: 5 True Class: 5
Test 183 Prediction: 0 True Class: 0
Test 184 Prediction: 2 True Class: 8
Test 185 Prediction: 9 True Class: 9
Test 186 Prediction: 2 True Class: 2
Test 187 Prediction: 5 True Class: 5
Test 188 Prediction: 0 True Class: 0
```

```
Test 189 Prediction: 1 True Class: 1
Test 190 Prediction: 1 True Class: 1
Test 191 Prediction: 1 True Class: 1
Test 192 Prediction: 0 True Class: 0
Test 193 Prediction: 4 True Class: 9
Test 194 Prediction: 0 True Class: 0
Test 195 Prediction: 1 True Class: 3
Test 196 Prediction: 1 True Class: 1
Test 197 Prediction: 6 True Class: 6
Test 198 Prediction: 4 True Class: 4
Test 199 Prediction: 2 True Class: 2
Done!
Accuracy: 0.92
```

K-Means Example

Implement K-Means algorithm with TensorFlow, and apply it to classify handwritten digit images. This example is using the MNIST database of handwritten digits as training samples (http://yann.lecun.com/exdb/mnist/).

Note: This example requires TensorFlow v1.1.0 or over.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

import numpy as np
import tensorflow as tf
from tensorflow.contrib.factorization import KMeans

# Ignore all GPUs, tf random forest does not benefit from it.
import os
os.environ["CUDA_VISIBLE_DEVICES"] = ""
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
full_data_x = mnist.train.images
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
num_steps = 50 # Total steps to train
batch_size = 1024 # The number of samples per batch
k = 25 # The number of clusters
num_classes = 10 # The 10 digits
num_features = 784 # Each image is 28x28 pixels

# Input images
X = tf.placeholder(tf.float32, shape=[None, num_features])
# Labels (for assigning a label to a centroid and testing)
Y = tf.placeholder(tf.float32, shape=[None, num_classes])

# K-Means Parameters
kmeans = KMeans(inputs=X, num_clusters=k, distance_metric='cosin
e',
                use_mini_batch=True)
```

```
# Build KMeans graph
(all_scores, cluster_idx, scores, cluster_centers_initialized,
 cluster_centers_vars,init_op,train_op) = kmeans.training_graph(
)
cluster_idx = cluster_idx[0] # fix for cluster_idx being a tuple
avg_distance = tf.reduce_mean(scores)

# Initialize the variables (i.e. assign their default value)
init_vars = tf.global_variables_initializer()
```

```
# Start TensorFlow session
sess = tf.Session()

# Run the initializer
sess.run(init_vars, feed_dict={X: full_data_x})
sess.run(init_op, feed_dict={X: full_data_x})

# Training
for i in range(1, num_steps + 1):
    _, d, idx = sess.run([train_op, avg_distance, cluster_idx],
                         feed_dict={X: full_data_x})
    if i % 10 == 0 or i == 1:
        print("Step %i, Avg Distance: %f" % (i, d))
```

```
Step 1, Avg Distance: 0.341471
Step 10, Avg Distance: 0.221609
Step 20, Avg Distance: 0.220328
Step 30, Avg Distance: 0.219776
Step 40, Avg Distance: 0.219419
Step 50, Avg Distance: 0.219154
```

```
# Assign a label to each centroid
# Count total number of labels per centroid, using the label of
each training
# sample to their closest centroid (given by 'idx')
counts = np.zeros(shape=(k, num_classes))
for i in range(len(idx)):
    counts[idx[i]] += mnist.train.labels[i]
# Assign the most frequent label to the centroid
labels_map = [np.argmax(c) for c in counts]
labels_map = tf.convert_to_tensor(labels_map)

# Evaluation ops
# Lookup: centroid_id -> label
cluster_label = tf.nn.embedding_lookup(labels_map, cluster_idx)
# Compute accuracy
correct_prediction = tf.equal(cluster_label, tf.cast(tf.argmax(Y
, 1), tf.int32))
accuracy_op = tf.reduce_mean(tf.cast(correct_prediction, tf.floa
t32))

# Test Model
test_x, test_y = mnist.test.images, mnist.test.labels
print("Test Accuracy:", sess.run(accuracy_op, feed_dict={X: test
_x, Y: test_y}))
```

```
Test Accuracy: 0.7127
```

Random Forest Example

Implement Random Forest algorithm with TensorFlow, and apply it to classify handwritten digit images. This example is using the MNIST database of handwritten digits as training samples (http://yann.lecun.com/exdb/mnist/).

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

import tensorflow as tf
from tensorflow.python.ops import resources
from tensorflow.contrib.tensor_forest.python import tensor_fores
t

# Ignore all GPUs, tf random forest does not benefit from it.
import os
os.environ["CUDA_VISIBLE_DEVICES"] = ""
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes
.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
num_steps = 500 # Total steps to train
batch_size = 1024 # The number of samples per batch
num_classes = 10 # The 10 digits
num_features = 784 # Each image is 28x28 pixels
num_trees = 10
max_nodes = 1000

# Input and Target data
X = tf.placeholder(tf.float32, shape=[None, num_features])
# For random forest, labels must be integers (the class id)
Y = tf.placeholder(tf.int32, shape=[None])

# Random Forest Parameters
hparams = tensor_forest.ForestHParams(num_classes=num_classes,
                                      num_features=num_features,
                                      num_trees=num_trees,
                                      max_nodes=max_nodes).fill(
)
```

```
# Build the Random Forest
forest_graph = tensor_forest.RandomForestGraphs(hparams)
# Get training graph and loss
train_op = forest_graph.training_graph(X, Y)
loss_op = forest_graph.training_loss(X, Y)

# Measure the accuracy
infer_op, _, _ = forest_graph.inference_graph(X)
correct_prediction = tf.equal(tf.argmax(infer_op, 1), tf.cast(Y,
 tf.int64))
accuracy_op = tf.reduce_mean(tf.cast(correct_prediction, tf.floa
t32))

# Initialize the variables (i.e. assign their default value) and
 forest resources
init_vars = tf.group(tf.global_variables_initializer(),
    resources.initialize_resources(resources.shared_resources())
)
```

```
INFO:tensorflow:Constructing forest with params =
INFO:tensorflow:{'valid_leaf_threshold': 1, 'split_after_samples
': 250, 'num_output_columns': 11, 'feature_bagging_fraction': 1.
0, 'split_initializations_per_input': 3, 'bagged_features': None
, 'min_split_samples': 5, 'max_nodes': 1000, 'num_features': 784
, 'num_trees': 10, 'num_splits_to_consider': 784, 'base_random_s
eed': 0, 'num_outputs': 1, 'dominate_fraction': 0.99, 'max_ferti
le_nodes': 500, 'bagged_num_features': 784, 'dominate_method': '
bootstrap', 'bagging_fraction': 1.0, 'regression': False, 'num_c
lasses': 10}
INFO:tensorflow:training graph for tree: 0
INFO:tensorflow:training graph for tree: 1
INFO:tensorflow:training graph for tree: 2
INFO:tensorflow:training graph for tree: 3
INFO:tensorflow:training graph for tree: 4
INFO:tensorflow:training graph for tree: 5
INFO:tensorflow:training graph for tree: 6
INFO:tensorflow:training graph for tree: 7
INFO:tensorflow:training graph for tree: 8
INFO:tensorflow:training graph for tree: 9
```

```
# Start TensorFlow session
sess = tf.train.MonitoredSession()

# Run the initializer
sess.run(init_vars)

# Training
for i in range(1, num_steps + 1):
    # Prepare Data
    # Get the next batch of MNIST data (only images are needed,
not labels)
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    _, l = sess.run([train_op, loss_op], feed_dict={X: batch_x,
Y: batch_y})
    if i % 50 == 0 or i == 1:
        acc = sess.run(accuracy_op, feed_dict={X: batch_x, Y: ba
tch_y})
        print('Step %i, Loss: %f, Acc: %f' % (i, l, acc))

# Test Model
test_x, test_y = mnist.test.images, mnist.test.labels
print("Test Accuracy:", sess.run(accuracy_op, feed_dict={X: test
_x, Y: test_y}))
```

```
Step 1, Loss: -0.000000, Acc: 0.112305
Step 50, Loss: -123.800003, Acc: 0.863281
Step 100, Loss: -274.200012, Acc: 0.863281
Step 150, Loss: -425.399994, Acc: 0.872070
Step 200, Loss: -582.799988, Acc: 0.917969
Step 250, Loss: -740.200012, Acc: 0.912109
Step 300, Loss: -895.799988, Acc: 0.939453
Step 350, Loss: -998.000000, Acc: 0.924805
Step 400, Loss: -998.000000, Acc: 0.940430
Step 450, Loss: -998.000000, Acc: 0.914062
Step 500, Loss: -998.000000, Acc: 0.927734
Test Accuracy: 0.9204
```

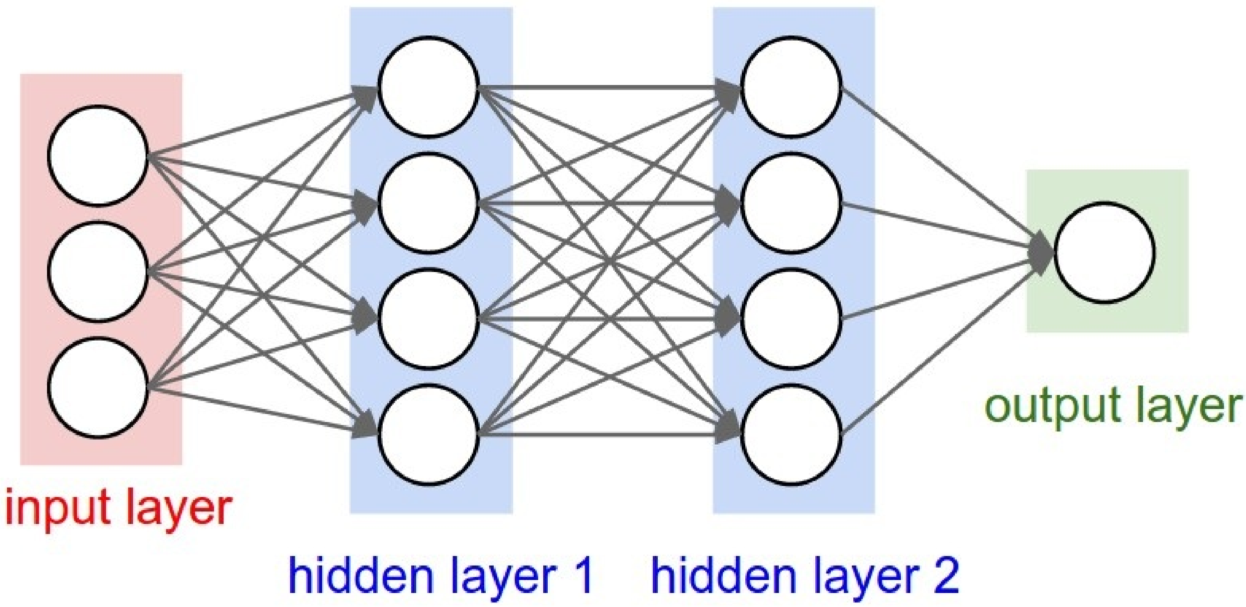
Neural Networks

Neural Network Example

Build a 2-hidden layers fully connected neural network (a.k.a multilayer perceptron) with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

Neural Network Overview



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

import tensorflow as tf
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.1
num_steps = 500
batch_size = 128
display_step = 100

# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1]))
,
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])
),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes
]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['
b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out'
]
    return out_layer
```

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict=
{X: batch_x,

 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

```
Step 1, Minibatch Loss= 13208.1406, Training Accuracy= 0.266
Step 100, Minibatch Loss= 462.8610, Training Accuracy= 0.867
Step 200, Minibatch Loss= 232.8298, Training Accuracy= 0.844
Step 300, Minibatch Loss= 85.2141, Training Accuracy= 0.891
Step 400, Minibatch Loss= 38.0552, Training Accuracy= 0.883
Step 500, Minibatch Loss= 55.3689, Training Accuracy= 0.867
Optimization Finished!
Testing Accuracy: 0.8729
```

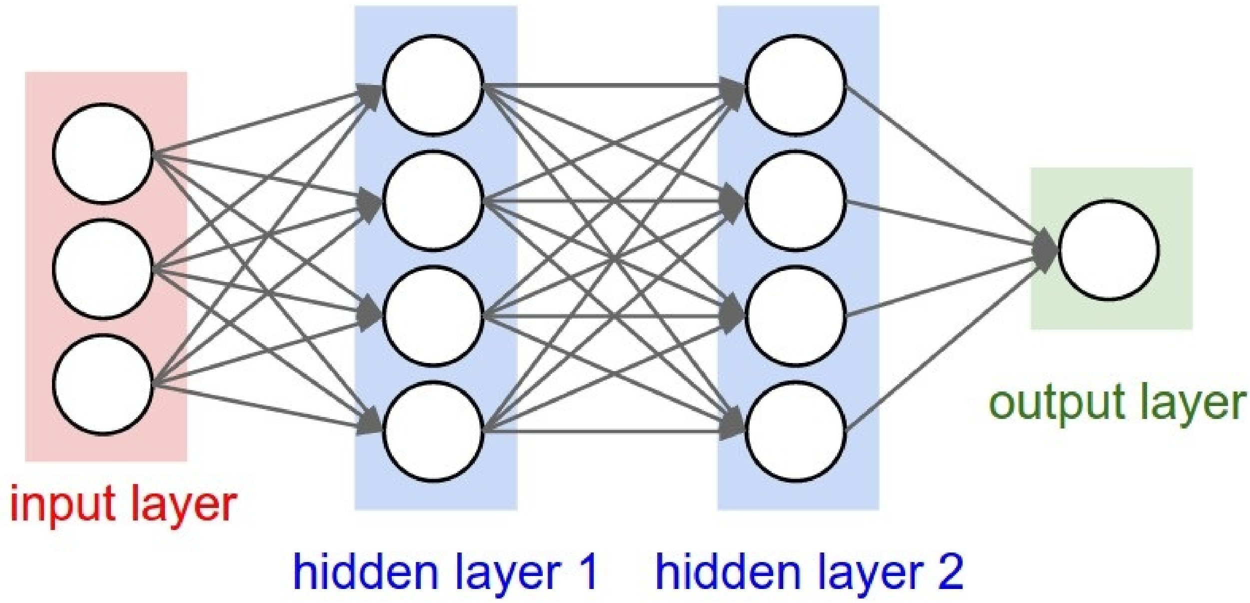
Neural Network Example

Build a 2-hidden layers fully connected neural network (a.k.a multilayer perceptron) with TensorFlow.

This example is using some of TensorFlow higher-level wrappers (tf.estimators, tf.layers, tf.metrics, ...), you can check 'neural_network_raw' example for a raw, and more detailed TensorFlow implementation.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

Neural Network Overview



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.1
num_steps = 1000
batch_size = 128
display_step = 100

# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
```

```
# Define the input function for training
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.train.images}, y=mnist.train.labels,
    batch_size=batch_size, num_epochs=None, shuffle=True)
```

```
# Define the neural network
def neural_net(x_dict):
    # TF Estimator input is a dict, in case of multiple inputs
    x = x_dict['images']
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.layers.dense(x, n_hidden_1)
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.layers.dense(layer_1, n_hidden_2)
    # Output fully connected layer with a neuron for each class
    out_layer = tf.layers.dense(layer_2, num_classes)
    return out_layer
```

```
# Define the model function (following TF Estimator Template)
def model_fn(features, labels, mode):

    # Build the neural network
    logits = neural_net(features)

    # Predictions
    pred_classes = tf.argmax(logits, axis=1)
    pred_probas = tf.nn.softmax(logits)

    # If prediction mode, early return
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode, predictions=pred
_classes)

    # Define loss and optimizer
    loss_op = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
with_logits(
        logits=logits, labels=tf.cast(labels, dtype=tf.int32)))
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=
learning_rate)
    train_op = optimizer.minimize(loss_op, global_step=tf.train.
get_global_step())

    # Evaluate the accuracy of the model
    acc_op = tf.metrics.accuracy(labels=labels, predictions=pred
_classes)

    # TF Estimators requires to return a EstimatorSpec, that spe
cify
    # the different ops for training, evaluating, ...
    estim_specs = tf.estimator.EstimatorSpec(
      mode=mode,
      predictions=pred_classes,
      loss=loss_op,
      train_op=train_op,
      eval_metric_ops={'accuracy': acc_op})

    return estim_specs
```

```
# Build the Estimator
model = tf.estimator.Estimator(model_fn)
```

```
INFO:tensorflow:Using default config.
WARNING:tensorflow:Using temporary folder as model directory: /t
mp/tmpu7vjLA
INFO:tensorflow:Using config: {'_save_checkpoints_secs': 600, '_
session_config': None, '_keep_checkpoint_max': 5, '_tf_random_se
ed': 1, '_keep_checkpoint_every_n_hours': 10000, '_log_step_coun
t_steps': 100, '_save_checkpoints_steps': None, '_model_dir': '/
tmp/tmpu7vjLA', '_save_summary_steps': 100}
```

```
# Train the Model
model.train(input_fn, steps=num_steps)
```

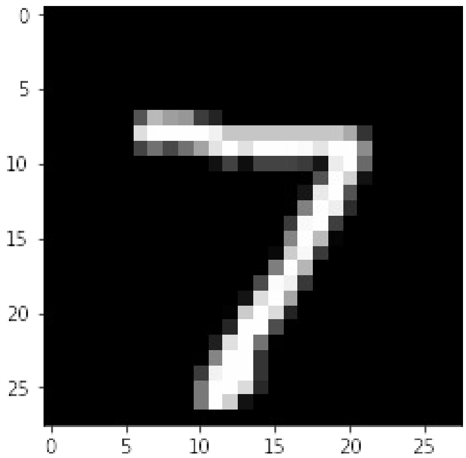
```
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Saving checkpoints for 1 into /tmp/tmpu7vjLA/mod
el.ckpt.
INFO:tensorflow:loss = 2.44919, step = 1
INFO:tensorflow:global_step/sec: 602.544
INFO:tensorflow:loss = 0.344767, step = 101 (0.167 sec)
INFO:tensorflow:global_step/sec: 618.839
INFO:tensorflow:loss = 0.277633, step = 201 (0.162 sec)
INFO:tensorflow:global_step/sec: 626.418
INFO:tensorflow:loss = 0.407796, step = 301 (0.160 sec)
INFO:tensorflow:global_step/sec: 624.765
INFO:tensorflow:loss = 0.376889, step = 401 (0.160 sec)
INFO:tensorflow:global_step/sec: 624.091
INFO:tensorflow:loss = 0.319697, step = 501 (0.160 sec)
INFO:tensorflow:global_step/sec: 616.907
INFO:tensorflow:loss = 0.39049, step = 601 (0.162 sec)
INFO:tensorflow:global_step/sec: 623.371
INFO:tensorflow:loss = 0.336831, step = 701 (0.161 sec)
INFO:tensorflow:global_step/sec: 617.429
INFO:tensorflow:loss = 0.312776, step = 801 (0.162 sec)
INFO:tensorflow:global_step/sec: 620.825
INFO:tensorflow:loss = 0.312817, step = 901 (0.161 sec)
INFO:tensorflow:Saving checkpoints for 1000 into /tmp/tmpu7vjLA/
model.ckpt.
INFO:tensorflow:Loss for final step: 0.24931.




<tensorflow.python.estimator.estimator.Estimator at 0x7f21ac5976
90>
```

```
# Evaluate the Model
# Define the input function for evaluating
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.test.images}, y=mnist.test.labels,
    batch_size=batch_size, shuffle=False)
# Use the Estimator 'evaluate' method
model.evaluate(input_fn)
```

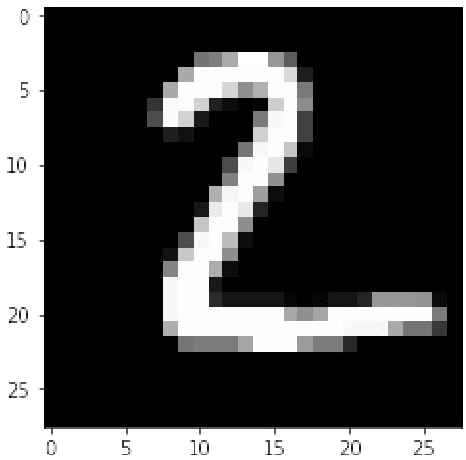
```
INFO:tensorflow:Starting evaluation at 2017-08-21-13:57:02
INFO:tensorflow:Restoring parameters from /tmp/tmpu7vjLA/model.c
kpt-1000
INFO:tensorflow:Finished evaluation at 2017-08-21-13:57:02
INFO:tensorflow:Saving dict for global step 1000: accuracy = 0.9
189, global_step = 1000, loss = 0.286567




{'accuracy': 0.91890001, 'global_step': 1000, 'loss': 0.28656715
}
```

```
# Predict single images
n_images = 4
# Get images from test set
test_images = mnist.test.images[:n_images]
# Prepare the input data
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': test_images}, shuffle=False)
# Use the model to predict the images class
preds = list(model.predict(input_fn))

# Display
for i in range(n_images):
    plt.imshow(np.reshape(test_images[i], [28, 28]), cmap='gray'
)
    plt.show()
    print("Model prediction:", preds[i])
```

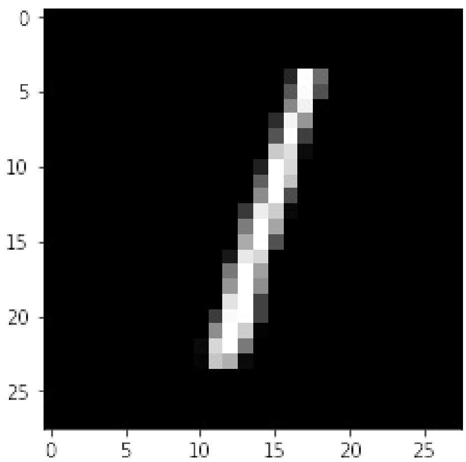
```
INFO:tensorflow:Restoring parameters from /tmp/tmpu7vjLA/model.c
kpt-1000
```



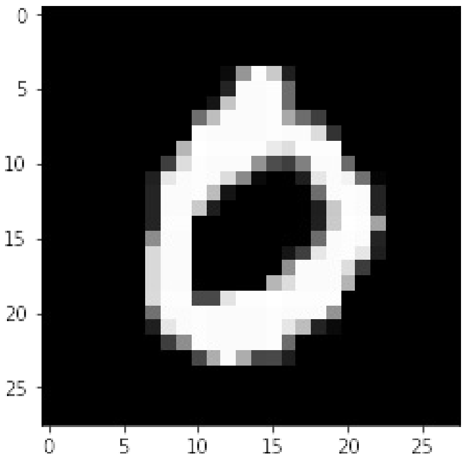

```
Model prediction: 7
```




```
Model prediction: 2
```




```
Model prediction: 1
```




```
Model prediction: 0
```

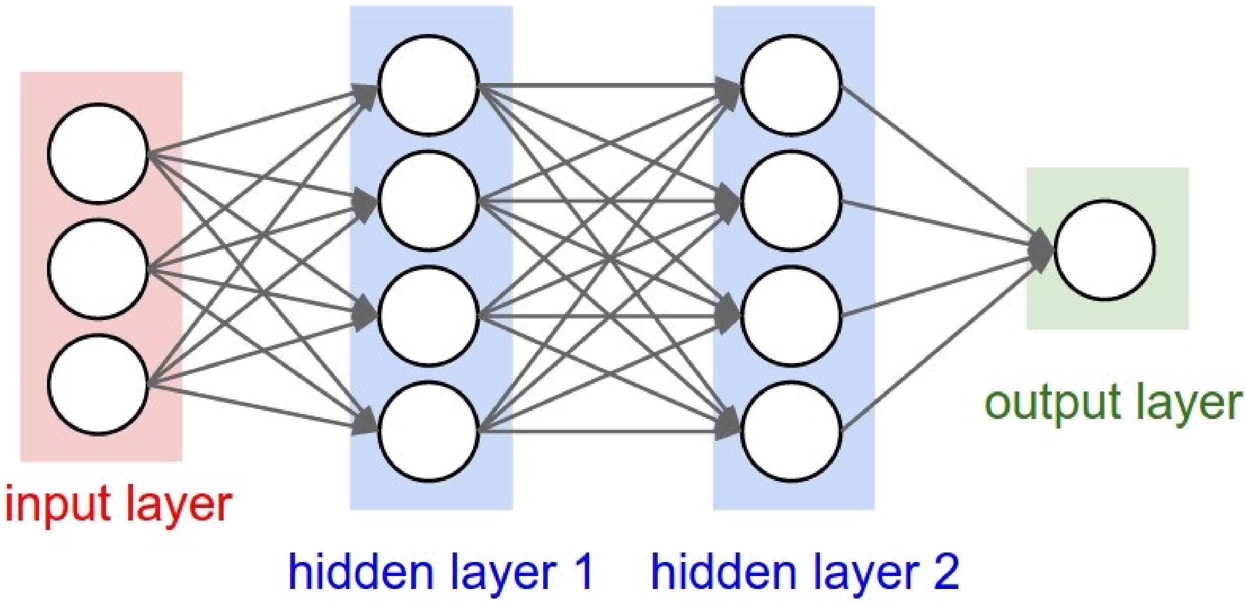
Neural Network with Eager API

Build a 2-hidden layers fully connected neural network (a.k.a multilayer perceptron) with TensorFlow's Eager API.

This example is using some of TensorFlow higher-level wrappers (tf.estimators, tf.layers, tf.metrics, ...), you can check 'neural_network_raw' example for a raw, and more detailed TensorFlow implementation.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

Neural Network Overview



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

import tensorflow as tf
import tensorflow.contrib.eager as tfe
```

```
# Set Eager API
tfe.enable_eager_execution()
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.001
num_steps = 1000
batch_size = 128
display_step = 100

# Network Parameters
n_hidden_1 = 256 # 1st layer number of neurons
n_hidden_2 = 256 # 2nd layer number of neurons
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
```

```
# Using TF Dataset to split data into batches
dataset = tf.data.Dataset.from_tensor_slices(
    (mnist.train.images, mnist.train.labels)).batch(batch_size)
dataset_iter = tfe.Iterator(dataset)
```

```
# Define the neural network. To use eager API and tf.layers API
together,
# we must instantiate a tfe.Network class as follow:
class NeuralNet(tfe.Network):
    def __init__(self):
        # Define each layer
        super(NeuralNet, self).__init__()
        # Hidden fully connected layer with 256 neurons
        self.layer1 = self.track_layer(
            tf.layers.Dense(n_hidden_1, activation=tf.nn.relu))
        # Hidden fully connected layer with 256 neurons
        self.layer2 = self.track_layer(
            tf.layers.Dense(n_hidden_2, activation=tf.nn.relu))
        # Output fully connected layer with a neuron for each cl
ass
        self.out_layer = self.track_layer(tf.layers.Dense(num_cl
asses))

    def call(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        return self.out_layer(x)


neural_net = NeuralNet()
```

```
# Cross-Entropy loss function
def loss_fn(inference_fn, inputs, labels):
    # Using sparse_softmax cross entropy
    return tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_wit
h_logits(
        logits=inference_fn(inputs), labels=labels))


# Calculate accuracy
def accuracy_fn(inference_fn, inputs, labels):
    prediction = tf.nn.softmax(inference_fn(inputs))
    correct_pred = tf.equal(tf.argmax(prediction, 1), labels)
    return tf.reduce_mean(tf.cast(correct_pred, tf.float32))


# SGD Optimizer
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)

# Compute gradients
grad = tfe.implicit_gradients(loss_fn)
```

```
# Training
average_loss = 0.
average_acc = 0.
for step in range(num_steps):

    # Iterate through the dataset
    try:
        d = dataset_iter.next()
    except StopIteration:
        # Refill queue
        dataset_iter = tfe.Iterator(dataset)
        d = dataset_iter.next()

    # Images
    x_batch = d[0]
    # Labels
    y_batch = tf.cast(d[1], dtype=tf.int64)

    # Compute the batch loss
    batch_loss = loss_fn(neural_net, x_batch, y_batch)
    average_loss += batch_loss
    # Compute the batch accuracy
    batch_accuracy = accuracy_fn(neural_net, x_batch, y_batch)
    average_acc += batch_accuracy

    if step == 0:
        # Display the initial cost, before optimizing
        print("Initial loss= {:.9f}".format(average_loss))

    # Update the variables following gradients info
    optimizer.apply_gradients(grad(neural_net, x_batch, y_batch)
)

    # Display info
    if (step + 1) % display_step == 0 or step == 0:
        if step > 0:
            average_loss /= display_step
            average_acc /= display_step
        print("Step:", '%04d' % (step + 1), " loss=",
              "{:.9f}".format(average_loss), " accuracy=",
              "{:.4f}".format(average_acc))
        average_loss = 0.
        average_acc = 0.
```

```
Initial loss= 2.340397596
Step: 0001  loss= 2.340397596  accuracy= 0.0703
Step: 0100  loss= 0.586046159  accuracy= 0.8305
Step: 0200  loss= 0.253318846  accuracy= 0.9282
Step: 0300  loss= 0.214748293  accuracy= 0.9377
Step: 0400  loss= 0.180644721  accuracy= 0.9466
Step: 0500  loss= 0.137285724  accuracy= 0.9591
Step: 0600  loss= 0.119845696  accuracy= 0.9636
Step: 0700  loss= 0.113618039  accuracy= 0.9665
Step: 0800  loss= 0.109642141  accuracy= 0.9676
Step: 0900  loss= 0.085067607  accuracy= 0.9746
Step: 1000  loss= 0.079819344  accuracy= 0.9754
```

```
# Evaluate model on the test image set
testX = mnist.test.images
testY = mnist.test.labels

test_acc = accuracy_fn(neural_net, testX, testY)
print("Testset Accuracy: {:.4f}".format(test_acc))
```

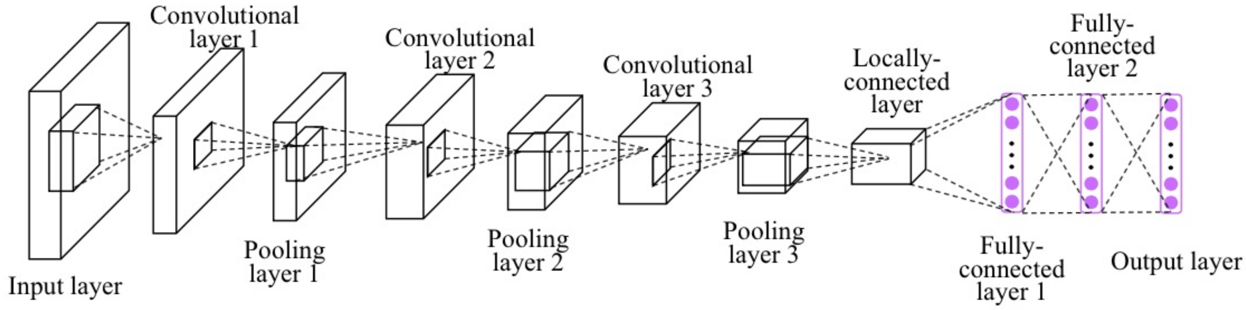
```
Testset Accuracy: 0.9719
```

Convolutional Neural Network Example

Build a convolutional neural network with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

CNN Overview



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Parameters
learning_rate = 0.001
num_steps = 500
batch_size = 128
display_step = 10

# Network Parameters
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
dropout = 0.75 # Dropout, probability to keep units

# tf Graph input
X = tf.placeholder(tf.float32, [None, num_input])
Y = tf.placeholder(tf.float32, [None, num_classes])
keep_prob = tf.placeholder(tf.float32) # dropout (keep probabili
ty)
```

```
# Create some wrappers for simplicity
def conv2d(x, W, b, strides=1):
    # Conv2D wrapper, with bias and relu activation
    x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], pad
ding='SAME')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)


def maxpool2d(x, k=2):
    # MaxPool2D wrapper
    return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k,
k, 1],
                          padding='SAME')


# Create model
def conv_net(x, weights, biases, dropout):
    # MNIST data input is a 1-D vector of 784 features (28*28 pi
xels)
    # Reshape to match picture format [Height x Width x Channel]
    # Tensor input become 4-D: [Batch Size, Height, Width, Chann
el]
    x = tf.reshape(x, shape=[-1, 28, 28, 1])

    # Convolution Layer
    conv1 = conv2d(x, weights['wc1'], biases['bc1'])
    # Max Pooling (down-sampling)
    conv1 = maxpool2d(conv1, k=2)

    # Convolution Layer
    conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
    # Max Pooling (down-sampling)
    conv2 = maxpool2d(conv2, k=2)

    # Fully connected layer
    # Reshape conv2 output to fit fully connected layer input
    fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_l
ist()[0]])
    fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
    fc1 = tf.nn.relu(fc1)
    # Apply Dropout
    fc1 = tf.nn.dropout(fc1, dropout)

    # Output, class prediction
    out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
    return out
```

```
# Store layers weight & bias
weights = {
    # 5x5 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
    # 5x5 conv, 32 inputs, 64 outputs
    'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
    # fully connected, 7*7*64 inputs, 1024 outputs
    'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
    # 1024 inputs, 10 outputs (class prediction)
    'out': tf.Variable(tf.random_normal([1024, num_classes]))
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}

# Construct model
logits = conv_net(X, weights, biases, keep_prob)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)


# Evaluate model
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1
))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y, ke
ep_prob: dropout})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict=
{X: batch_x,

 Y: batch_y,

 keep_prob: 1.0})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 256 MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images[:256]
,
                                      Y: mnist.test.labels[:256]
,
                                      keep_prob: 1.0}))
```

```
Step 1, Minibatch Loss= 63763.3047, Training Accuracy= 0.141
Step 10, Minibatch Loss= 26429.6680, Training Accuracy= 0.242
Step 20, Minibatch Loss= 12171.8584, Training Accuracy= 0.586
Step 30, Minibatch Loss= 6306.6318, Training Accuracy= 0.734
Step 40, Minibatch Loss= 5113.7583, Training Accuracy= 0.711
Step 50, Minibatch Loss= 4022.2131, Training Accuracy= 0.805
Step 60, Minibatch Loss= 3125.4949, Training Accuracy= 0.867
Step 70, Minibatch Loss= 2225.4875, Training Accuracy= 0.875
Step 80, Minibatch Loss= 1843.3540, Training Accuracy= 0.867
Step 90, Minibatch Loss= 1715.7744, Training Accuracy= 0.875
Step 100, Minibatch Loss= 2611.2708, Training Accuracy= 0.906
Step 110, Minibatch Loss= 4804.0913, Training Accuracy= 0.875
Step 120, Minibatch Loss= 1067.5258, Training Accuracy= 0.938
Step 130, Minibatch Loss= 2519.1514, Training Accuracy= 0.898
Step 140, Minibatch Loss= 2687.9292, Training Accuracy= 0.906
Step 150, Minibatch Loss= 1983.4077, Training Accuracy= 0.938
```

```
Step 160, Minibatch Loss= 2844.6553, Training Accuracy= 0.930
Step 170, Minibatch Loss= 3602.2524, Training Accuracy= 0.914
Step 180, Minibatch Loss= 175.3922, Training Accuracy= 0.961
Step 190, Minibatch Loss= 645.1918, Training Accuracy= 0.945
Step 200, Minibatch Loss= 1147.6567, Training Accuracy= 0.938
Step 210, Minibatch Loss= 1140.4148, Training Accuracy= 0.914
Step 220, Minibatch Loss= 1572.8756, Training Accuracy= 0.906
Step 230, Minibatch Loss= 1292.9274, Training Accuracy= 0.898
Step 240, Minibatch Loss= 1501.4623, Training Accuracy= 0.953
Step 250, Minibatch Loss= 1908.2997, Training Accuracy= 0.898
Step 260, Minibatch Loss= 2182.2380, Training Accuracy= 0.898
Step 270, Minibatch Loss= 487.5807, Training Accuracy= 0.961
Step 280, Minibatch Loss= 1284.1130, Training Accuracy= 0.945
Step 290, Minibatch Loss= 1232.4919, Training Accuracy= 0.891
Step 300, Minibatch Loss= 1198.8336, Training Accuracy= 0.945
Step 310, Minibatch Loss= 2010.5345, Training Accuracy= 0.906
Step 320, Minibatch Loss= 786.3917, Training Accuracy= 0.945
Step 330, Minibatch Loss= 1408.3556, Training Accuracy= 0.898
Step 340, Minibatch Loss= 1453.7538, Training Accuracy= 0.953
Step 350, Minibatch Loss= 999.8901, Training Accuracy= 0.906
Step 360, Minibatch Loss= 914.3958, Training Accuracy= 0.961
Step 370, Minibatch Loss= 488.0052, Training Accuracy= 0.938
Step 380, Minibatch Loss= 1070.8710, Training Accuracy= 0.922
Step 390, Minibatch Loss= 151.4658, Training Accuracy= 0.961
Step 400, Minibatch Loss= 555.3539, Training Accuracy= 0.953
Step 410, Minibatch Loss= 765.5746, Training Accuracy= 0.945
Step 420, Minibatch Loss= 326.9393, Training Accuracy= 0.969
Step 430, Minibatch Loss= 530.8968, Training Accuracy= 0.977
Step 440, Minibatch Loss= 463.3909, Training Accuracy= 0.977
Step 450, Minibatch Loss= 362.2226, Training Accuracy= 0.977
Step 460, Minibatch Loss= 414.0034, Training Accuracy= 0.953
Step 470, Minibatch Loss= 583.4587, Training Accuracy= 0.945
Step 480, Minibatch Loss= 566.1262, Training Accuracy= 0.969
Step 490, Minibatch Loss= 691.1143, Training Accuracy= 0.961
Step 500, Minibatch Loss= 282.8893, Training Accuracy= 0.984
Optimization Finished!
Testing Accuracy: 0.976562
```

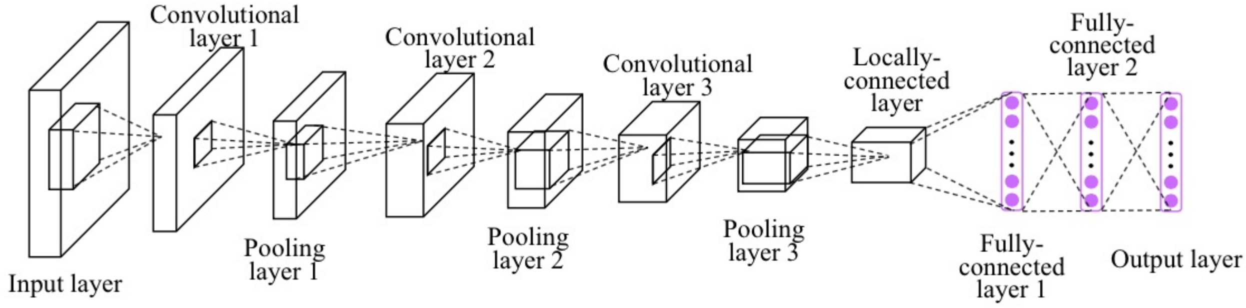
Convolutional Neural Network Example

Build a convolutional neural network with TensorFlow.

This example is using TensorFlow layers API, see 'convolutional_network_raw' example for a raw TensorFlow implementation with variables.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

CNN Overview



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=False)

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Parameters
learning_rate = 0.001
num_steps = 2000
batch_size = 128

# Network Parameters
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
dropout = 0.25 # Dropout, probability to drop a unit
```

```
# Create the neural network
def conv_net(x_dict, n_classes, dropout, reuse, is_training):

    # Define a scope for reusing the variables
    with tf.variable_scope('ConvNet', reuse=reuse):
        # TF Estimator input is a dict, in case of multiple inputs
        x = x_dict['images']

        # MNIST data input is a 1-D vector of 784 features (28*28 pixels)
        # Reshape to match picture format [Height x Width x Channel]
        # Tensor input become 4-D: [Batch Size, Height, Width, Channel]
        x = tf.reshape(x, shape=[-1, 28, 28, 1])

        # Convolution Layer with 32 filters and a kernel size of 5
        conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu)
        # Max Pooling (down-sampling) with strides of 2 and kernel size of 2
        conv1 = tf.layers.max_pooling2d(conv1, 2, 2)

        # Convolution Layer with 64 filters and a kernel size of 3
        conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.relu)
        # Max Pooling (down-sampling) with strides of 2 and kernel size of 2
        conv2 = tf.layers.max_pooling2d(conv2, 2, 2)

        # Flatten the data to a 1-D vector for the fully connected layer
        fc1 = tf.contrib.layers.flatten(conv2)

        # Fully connected layer (in tf contrib folder for now)
        fc1 = tf.layers.dense(fc1, 1024)
        # Apply Dropout (if is_training is False, dropout is not applied)
        fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_training)

        # Output layer, class prediction
        out = tf.layers.dense(fc1, n_classes)

    return out
```

```
# Define the model function (following TF Estimator Template)
def model_fn(features, labels, mode):

    # Build the neural network
    # Because Dropout have different behavior at training and pr
ediction time, we
    # need to create 2 distinct computation graphs that still sh
are the same weights.
    logits_train = conv_net(features, num_classes, dropout, reus
e=False, is_training=True)
    logits_test = conv_net(features, num_classes, dropout, reuse=
True, is_training=False)

    # Predictions
    pred_classes = tf.argmax(logits_test, axis=1)
    pred_probas = tf.nn.softmax(logits_test)

    # If prediction mode, early return
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode, predictions=pred
_classes)

    # Define loss and optimizer
    loss_op = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_
with_logits(
        logits=logits_train, labels=tf.cast(labels, dtype=tf.int
32)))
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_ra
te)
    train_op = optimizer.minimize(loss_op, global_step=tf.train.
get_global_step())

    # Evaluate the accuracy of the model
    acc_op = tf.metrics.accuracy(labels=labels, predictions=pred
_classes)

    # TF Estimators requires to return a EstimatorSpec, that spe
cify
    # the different ops for training, evaluating, ...
    estim_specs = tf.estimator.EstimatorSpec(
      mode=mode,
      predictions=pred_classes,
      loss=loss_op,
      train_op=train_op,
      eval_metric_ops={'accuracy': acc_op})

    return estim_specs
```

```
# Build the Estimator
model = tf.estimator.Estimator(model_fn)
```

```
INFO:tensorflow:Using default config.
WARNING:tensorflow:Using temporary folder as model directory: /t
mp/tmpdhd6F4
INFO:tensorflow:Using config: {'_save_checkpoints_secs': 600, '_
session_config': None, '_keep_checkpoint_max': 5, '_tf_random_se
ed': 1, '_keep_checkpoint_every_n_hours': 10000, '_log_step_coun
t_steps': 100, '_save_checkpoints_steps': None, '_model_dir': '/
tmp/tmpdhd6F4', '_save_summary_steps': 100}
```

```
# Define the input function for training
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.train.images}, y=mnist.train.labels,
    batch_size=batch_size, num_epochs=None, shuffle=True)
# Train the Model
model.train(input_fn, steps=num_steps)
```

```
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Saving checkpoints for 1 into /tmp/tmpdhd6F4/mod
el.ckpt.
INFO:tensorflow:loss = 2.39026, step = 1
INFO:tensorflow:global_step/sec: 238.314
INFO:tensorflow:loss = 0.237997, step = 101 (0.421 sec)
INFO:tensorflow:global_step/sec: 255.312
INFO:tensorflow:loss = 0.0954537, step = 201 (0.392 sec)
INFO:tensorflow:global_step/sec: 257.194
INFO:tensorflow:loss = 0.121477, step = 301 (0.389 sec)
INFO:tensorflow:global_step/sec: 255.018
INFO:tensorflow:loss = 0.0539927, step = 401 (0.392 sec)
INFO:tensorflow:global_step/sec: 254.293
INFO:tensorflow:loss = 0.0440369, step = 501 (0.393 sec)
INFO:tensorflow:global_step/sec: 256.501
INFO:tensorflow:loss = 0.0247431, step = 601 (0.390 sec)
INFO:tensorflow:global_step/sec: 252.956
INFO:tensorflow:loss = 0.0738082, step = 701 (0.395 sec)
INFO:tensorflow:global_step/sec: 253.222
INFO:tensorflow:loss = 0.134998, step = 801 (0.395 sec)
INFO:tensorflow:global_step/sec: 255.606
INFO:tensorflow:loss = 0.00438448, step = 901 (0.391 sec)
INFO:tensorflow:global_step/sec: 256.306
INFO:tensorflow:loss = 0.0471991, step = 1001 (0.390 sec)
INFO:tensorflow:global_step/sec: 255.352
INFO:tensorflow:loss = 0.0371172, step = 1101 (0.392 sec)
INFO:tensorflow:global_step/sec: 253.277
INFO:tensorflow:loss = 0.0129522, step = 1201 (0.395 sec)
```

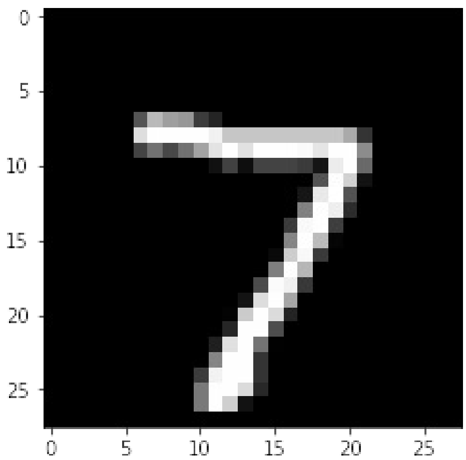
```
INFO:tensorflow:global_step/sec: 252.49
INFO:tensorflow:loss = 0.039862, step = 1301 (0.396 sec)
INFO:tensorflow:global_step/sec: 253.902
INFO:tensorflow:loss = 0.0520571, step = 1401 (0.394 sec)
INFO:tensorflow:global_step/sec: 255.572
INFO:tensorflow:loss = 0.0307549, step = 1501 (0.392 sec)
INFO:tensorflow:global_step/sec: 254.32
INFO:tensorflow:loss = 0.0108862, step = 1601 (0.393 sec)
INFO:tensorflow:global_step/sec: 255.62
INFO:tensorflow:loss = 0.0294434, step = 1701 (0.391 sec)
INFO:tensorflow:global_step/sec: 254.349
INFO:tensorflow:loss = 0.0179781, step = 1801 (0.393 sec)
INFO:tensorflow:global_step/sec: 255.508
INFO:tensorflow:loss = 0.0375271, step = 1901 (0.391 sec)
INFO:tensorflow:Saving checkpoints for 2000 into /tmp/tmpdhd6F4/
model.ckpt.
INFO:tensorflow:Loss for final step: 0.00440777.





<tensorflow.python.estimator.estimator.Estimator at 0x7fb80ca55c
90>
```

```
# Evaluate the Model
# Define the input function for evaluating
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': mnist.test.images}, y=mnist.test.labels,
    batch_size=batch_size, shuffle=False)
# Use the Estimator 'evaluate' method
model.evaluate(input_fn)
```

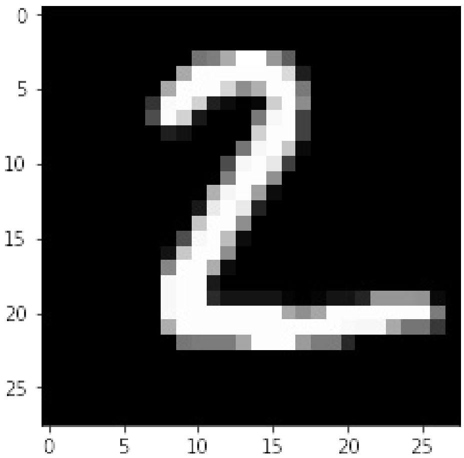
```
INFO:tensorflow:Starting evaluation at 2017-08-21-14:25:29
INFO:tensorflow:Restoring parameters from /tmp/tmpdhd6F4/model.c
kpt-2000
INFO:tensorflow:Finished evaluation at 2017-08-21-14:25:29
INFO:tensorflow:Saving dict for global step 2000: accuracy = 0.9
908, global_step = 2000, loss = 0.0382241





{'accuracy': 0.99080002, 'global_step': 2000, 'loss': 0.03822408
6}
```

```
# Predict single images
n_images = 4
# Get images from test set
test_images = mnist.test.images[:n_images]
# Prepare the input data
input_fn = tf.estimator.inputs.numpy_input_fn(
    x={'images': test_images}, shuffle=False)
# Use the model to predict the images class
preds = list(model.predict(input_fn))

# Display
for i in range(n_images):
    plt.imshow(np.reshape(test_images[i], [28, 28]), cmap='gray'
)
    plt.show()
    print("Model prediction:", preds[i])
```

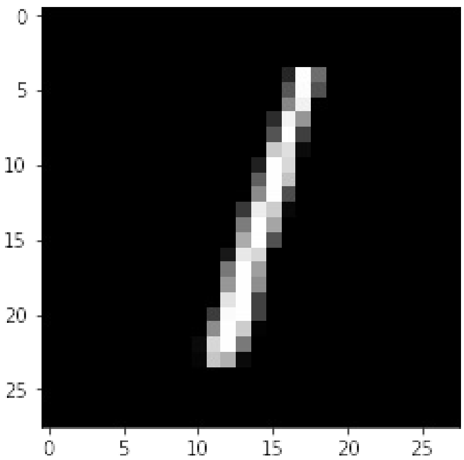
```
INFO:tensorflow:Restoring parameters from /tmp/tmpdhd6F4/model.c
kpt-2000
```



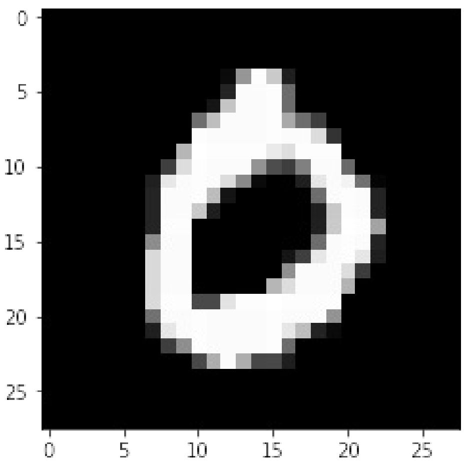

```
Model prediction: 7
```



```
Model prediction: 2
```



```
Model prediction: 1
```



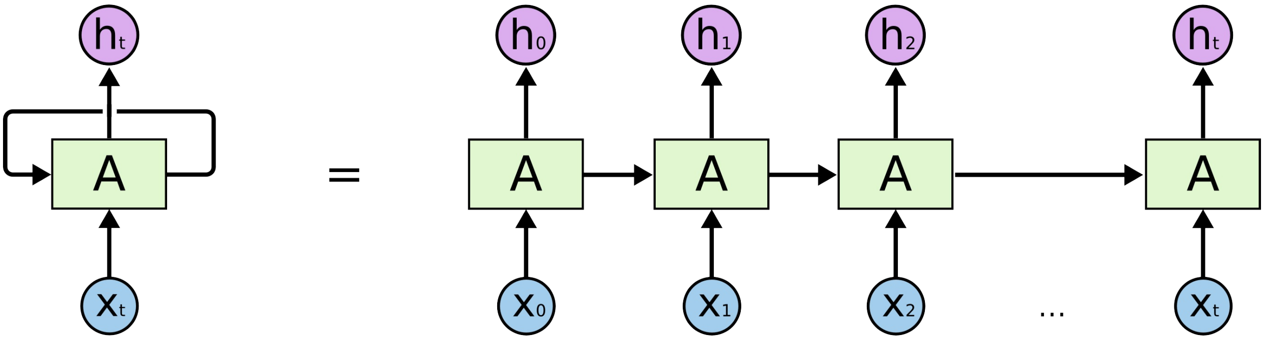
```
Model prediction: 0
```

Recurrent Neural Network Example

Build a recurrent neural network (LSTM) with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

RNN Overview



References:

- Long Short Term Memory, Sepp Hochreiter & Jurgen Schmidhuber, Neural Computation 9(8): 1735-1780, 1997.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

To classify images using a recurrent neural network, we consider every image row as a sequence of pixels. Because MNIST image shape is 28*28px, we will then handle 28 sequences of 28 timesteps for every sample.

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

import tensorflow as tf
from tensorflow.contrib import rnn

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Parameters
learning_rate = 0.001
training_steps = 10000
batch_size = 128
display_step = 200

# Network Parameters
num_input = 28 # MNIST data input (img shape: 28*28)
timesteps = 28 # timesteps
num_hidden = 128 # hidden layer num of features
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, timesteps, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Define weights
weights = {
    'out': tf.Variable(tf.random_normal([num_hidden, num_classes
]))
}
biases = {
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```
def RNN(x, weights, biases):

    # Prepare data shape to match `rnn` function requirements
    # Current data input shape: (batch_size, timesteps, n_input)
    # Required shape: 'timesteps' tensors list of shape (batch_s
ize, n_input)

    # Unstack to get a list of 'timesteps' tensors of shape (bat
ch_size, n_input)
    x = tf.unstack(x, timesteps, 1)

    # Define a lstm cell with tensorflow
    lstm_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0)

    # Get lstm cell output
    outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.floa
t32)

    # Linear activation, using rnn inner loop last output
    return tf.matmul(outputs[-1], weights['out']) + biases['out'
]
```

```
logits = RNN(X, weights, biases)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
(
    logits=logits, labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lear
ning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1
))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, training_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Reshape data to get 28 seq of 28 elements
        batch_x = batch_x.reshape((batch_size, timesteps, num_in
put))
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict=
{X: batch_x,

 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 128 mnist test images
    test_len = 128
    test_data = mnist.test.images[:test_len].reshape((-1, timest
eps, num_input))
    test_label = mnist.test.labels[:test_len]
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: test_data, Y: test_labe
l}))
```

```
Step 1, Minibatch Loss= 2.6268, Training Accuracy= 0.102
Step 200, Minibatch Loss= 2.0722, Training Accuracy= 0.328
Step 400, Minibatch Loss= 1.9181, Training Accuracy= 0.336
Step 600, Minibatch Loss= 1.8858, Training Accuracy= 0.336
Step 800, Minibatch Loss= 1.7022, Training Accuracy= 0.422
Step 1000, Minibatch Loss= 1.6365, Training Accuracy= 0.477
Step 1200, Minibatch Loss= 1.6691, Training Accuracy= 0.516
Step 1400, Minibatch Loss= 1.4626, Training Accuracy= 0.547
Step 1600, Minibatch Loss= 1.4707, Training Accuracy= 0.539
Step 1800, Minibatch Loss= 1.4087, Training Accuracy= 0.570
Step 2000, Minibatch Loss= 1.3033, Training Accuracy= 0.570
Step 2200, Minibatch Loss= 1.3773, Training Accuracy= 0.508
Step 2400, Minibatch Loss= 1.3092, Training Accuracy= 0.570
Step 2600, Minibatch Loss= 1.2272, Training Accuracy= 0.609
Step 2800, Minibatch Loss= 1.1827, Training Accuracy= 0.633
```

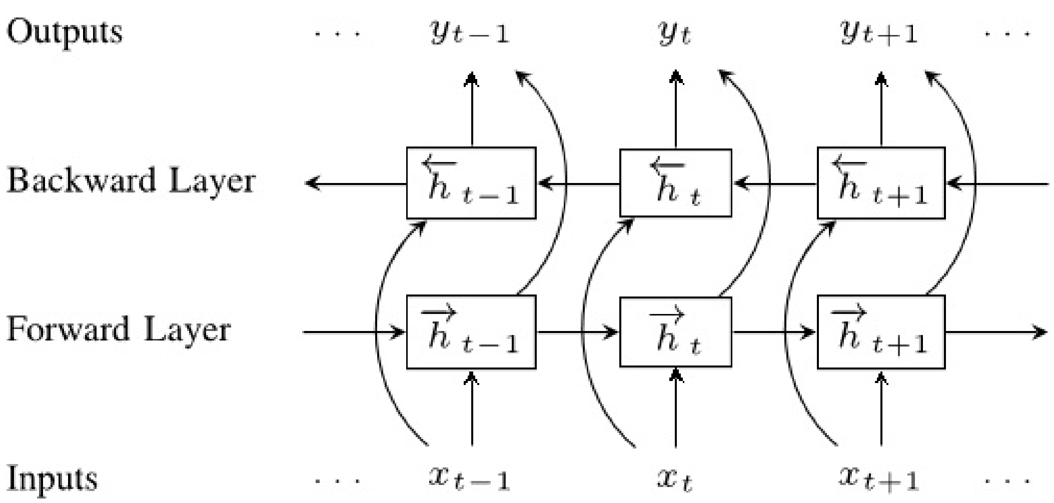
```
Step 3000, Minibatch Loss= 1.0453, Training Accuracy= 0.641
Step 3200, Minibatch Loss= 1.0400, Training Accuracy= 0.648
Step 3400, Minibatch Loss= 1.1145, Training Accuracy= 0.656
Step 3600, Minibatch Loss= 0.9884, Training Accuracy= 0.688
Step 3800, Minibatch Loss= 1.0395, Training Accuracy= 0.703
Step 4000, Minibatch Loss= 1.0096, Training Accuracy= 0.664
Step 4200, Minibatch Loss= 0.8806, Training Accuracy= 0.758
Step 4400, Minibatch Loss= 0.9090, Training Accuracy= 0.766
Step 4600, Minibatch Loss= 1.0060, Training Accuracy= 0.703
Step 4800, Minibatch Loss= 0.8954, Training Accuracy= 0.703
Step 5000, Minibatch Loss= 0.8163, Training Accuracy= 0.750
Step 5200, Minibatch Loss= 0.7620, Training Accuracy= 0.773
Step 5400, Minibatch Loss= 0.7388, Training Accuracy= 0.758
Step 5600, Minibatch Loss= 0.7604, Training Accuracy= 0.695
Step 5800, Minibatch Loss= 0.7459, Training Accuracy= 0.734
Step 6000, Minibatch Loss= 0.7448, Training Accuracy= 0.734
Step 6200, Minibatch Loss= 0.7208, Training Accuracy= 0.773
Step 6400, Minibatch Loss= 0.6557, Training Accuracy= 0.773
Step 6600, Minibatch Loss= 0.8616, Training Accuracy= 0.758
Step 6800, Minibatch Loss= 0.6089, Training Accuracy= 0.773
Step 7000, Minibatch Loss= 0.5020, Training Accuracy= 0.844
Step 7200, Minibatch Loss= 0.5980, Training Accuracy= 0.812
Step 7400, Minibatch Loss= 0.6786, Training Accuracy= 0.766
Step 7600, Minibatch Loss= 0.4891, Training Accuracy= 0.859
Step 7800, Minibatch Loss= 0.7042, Training Accuracy= 0.797
Step 8000, Minibatch Loss= 0.4200, Training Accuracy= 0.859
Step 8200, Minibatch Loss= 0.6442, Training Accuracy= 0.742
Step 8400, Minibatch Loss= 0.5569, Training Accuracy= 0.828
Step 8600, Minibatch Loss= 0.5838, Training Accuracy= 0.836
Step 8800, Minibatch Loss= 0.5579, Training Accuracy= 0.812
Step 9000, Minibatch Loss= 0.4337, Training Accuracy= 0.867
Step 9200, Minibatch Loss= 0.4366, Training Accuracy= 0.844
Step 9400, Minibatch Loss= 0.5051, Training Accuracy= 0.844
Step 9600, Minibatch Loss= 0.5244, Training Accuracy= 0.805
Step 9800, Minibatch Loss= 0.4932, Training Accuracy= 0.805
Step 10000, Minibatch Loss= 0.4833, Training Accuracy= 0.852
Optimization Finished!
Testing Accuracy: 0.882812
```

Bi-directional Recurrent Neural Network Example

Build a bi-directional recurrent neural network (LSTM) with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

BiRNN Overview



References:

- Long Short Term Memory, Sepp Hochreiter & Jurgen Schmidhuber, Neural Computation 9(8): 1735-1780, 1997.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

To classify images using a recurrent neural network, we consider every image row as a sequence of pixels. Because MNIST image shape is 28*28px, we will then handle 28 sequences of 28 timesteps for every sample.

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Parameters
learning_rate = 0.001
training_steps = 10000
batch_size = 128
display_step = 200

# Network Parameters
num_input = 28 # MNIST data input (img shape: 28*28)
timesteps = 28 # timesteps
num_hidden = 128 # hidden layer num of features
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, timesteps, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Define weights
weights = {
    # Hidden layer weights => 2*n_hidden because of forward + ba
ckward cells
    'out': tf.Variable(tf.random_normal([2*num_hidden, num_class
es]))
}
biases = {
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```
def BiRNN(x, weights, biases):

    # Prepare data shape to match `rnn` function requirements
    # Current data input shape: (batch_size, timesteps, n_input)
    # Required shape: 'timesteps' tensors list of shape (batch_s
ize, num_input)

    # Unstack to get a list of 'timesteps' tensors of shape (bat
ch_size, num_input)
    x = tf.unstack(x, timesteps, 1)

    # Define lstm cells with tensorflow
    # Forward direction cell
    lstm_fw_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0
)
    # Backward direction cell
    lstm_bw_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0
)

    # Get lstm cell output
    try:
        outputs, _, _ = rnn.static_bidirectional_rnn(lstm_fw_cel
l, lstm_bw_cell, x,
                                              dtype=tf.float32)
    except Exception: # Old TensorFlow version only returns outp
uts not states
        outputs = rnn.static_bidirectional_rnn(lstm_fw_cell, lst
m_bw_cell, x,
                                        dtype=tf.float32)

    # Linear activation, using rnn inner loop last output
    return tf.matmul(outputs[-1], weights['out']) + biases['out'
]
```

```
logits = BiRNN(X, weights, biases)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
(
    logits=logits, labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lear
ning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1
))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, training_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Reshape data to get 28 seq of 28 elements
        batch_x = batch_x.reshape((batch_size, timesteps, num_in
put))
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict=
{X: batch_x,

 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 128 mnist test images
    test_len = 128
    test_data = mnist.test.images[:test_len].reshape((-1, timest
eps, num_input))
    test_label = mnist.test.labels[:test_len]
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: test_data, Y: test_labe
l}))
```

```
Step 1, Minibatch Loss= 2.6218, Training Accuracy= 0.086
Step 200, Minibatch Loss= 2.1900, Training Accuracy= 0.211
Step 400, Minibatch Loss= 2.0144, Training Accuracy= 0.375
Step 600, Minibatch Loss= 1.8729, Training Accuracy= 0.445
Step 800, Minibatch Loss= 1.8000, Training Accuracy= 0.469
Step 1000, Minibatch Loss= 1.7244, Training Accuracy= 0.453
Step 1200, Minibatch Loss= 1.5657, Training Accuracy= 0.523
Step 1400, Minibatch Loss= 1.5473, Training Accuracy= 0.547
Step 1600, Minibatch Loss= 1.5288, Training Accuracy= 0.500
Step 1800, Minibatch Loss= 1.4203, Training Accuracy= 0.555
Step 2000, Minibatch Loss= 1.2525, Training Accuracy= 0.641
Step 2200, Minibatch Loss= 1.2696, Training Accuracy= 0.594
Step 2400, Minibatch Loss= 1.2000, Training Accuracy= 0.664
Step 2600, Minibatch Loss= 1.1017, Training Accuracy= 0.625
Step 2800, Minibatch Loss= 1.2656, Training Accuracy= 0.578
```

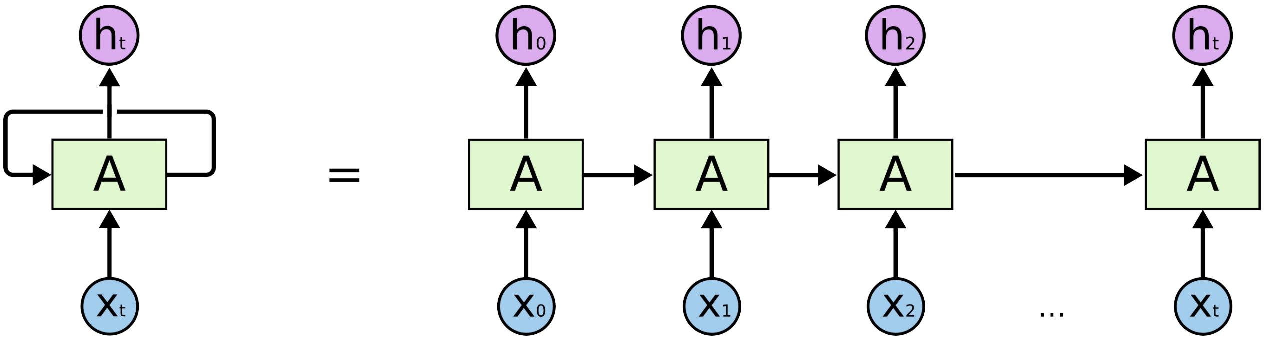
```
Step 3000, Minibatch Loss= 1.0830, Training Accuracy= 0.656
Step 3200, Minibatch Loss= 1.1522, Training Accuracy= 0.633
Step 3400, Minibatch Loss= 0.9484, Training Accuracy= 0.680
Step 3600, Minibatch Loss= 1.0470, Training Accuracy= 0.641
Step 3800, Minibatch Loss= 1.0609, Training Accuracy= 0.586
Step 4000, Minibatch Loss= 1.1853, Training Accuracy= 0.648
Step 4200, Minibatch Loss= 0.9438, Training Accuracy= 0.750
Step 4400, Minibatch Loss= 0.7986, Training Accuracy= 0.766
Step 4600, Minibatch Loss= 0.8070, Training Accuracy= 0.750
Step 4800, Minibatch Loss= 0.8382, Training Accuracy= 0.734
Step 5000, Minibatch Loss= 0.7397, Training Accuracy= 0.766
Step 5200, Minibatch Loss= 0.7870, Training Accuracy= 0.727
Step 5400, Minibatch Loss= 0.6380, Training Accuracy= 0.828
Step 5600, Minibatch Loss= 0.7975, Training Accuracy= 0.719
Step 5800, Minibatch Loss= 0.7934, Training Accuracy= 0.766
Step 6000, Minibatch Loss= 0.6628, Training Accuracy= 0.805
Step 6200, Minibatch Loss= 0.7958, Training Accuracy= 0.672
Step 6400, Minibatch Loss= 0.6582, Training Accuracy= 0.773
Step 6600, Minibatch Loss= 0.5908, Training Accuracy= 0.812
Step 6800, Minibatch Loss= 0.6182, Training Accuracy= 0.820
Step 7000, Minibatch Loss= 0.5513, Training Accuracy= 0.812
Step 7200, Minibatch Loss= 0.6683, Training Accuracy= 0.789
Step 7400, Minibatch Loss= 0.5337, Training Accuracy= 0.828
Step 7600, Minibatch Loss= 0.6428, Training Accuracy= 0.805
Step 7800, Minibatch Loss= 0.6708, Training Accuracy= 0.797
Step 8000, Minibatch Loss= 0.4664, Training Accuracy= 0.852
Step 8200, Minibatch Loss= 0.4249, Training Accuracy= 0.859
Step 8400, Minibatch Loss= 0.7723, Training Accuracy= 0.773
Step 8600, Minibatch Loss= 0.4706, Training Accuracy= 0.859
Step 8800, Minibatch Loss= 0.4800, Training Accuracy= 0.867
Step 9000, Minibatch Loss= 0.4636, Training Accuracy= 0.891
Step 9200, Minibatch Loss= 0.5734, Training Accuracy= 0.828
Step 9400, Minibatch Loss= 0.5548, Training Accuracy= 0.875
Step 9600, Minibatch Loss= 0.3575, Training Accuracy= 0.922
Step 9800, Minibatch Loss= 0.4566, Training Accuracy= 0.844
Step 10000, Minibatch Loss= 0.5125, Training Accuracy= 0.844
Optimization Finished!
Testing Accuracy: 0.890625
```

Dynamic Recurrent Neural Network.

TensorFlow implementation of a Recurrent Neural Network (LSTM) that performs dynamic computation over sequences with variable length. This example is using a toy dataset to classify linear sequences. The generated sequences have variable length.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

RNN Overview



References:

- Long Short Term Memory, Sepp Hochreiter & Jurgen Schmidhuber, Neural Computation 9(8): 1735-1780, 1997.

```
from __future__ import print_function

import tensorflow as tf
import random
```

```
# ====================
#  TOY DATA GENERATOR
# ====================

class ToySequenceData(object):
    """ Generate sequence of data with dynamic length.
    This class generate samples for training:
    - Class 0: linear sequences (i.e. [0, 1, 2, 3,...])
    - Class 1: random sequences (i.e. [1, 3, 10, 7,...])

    NOTICE:
    We have to pad each sequence to reach 'max_seq_len' for Tens
orFlow
    consistency (we cannot feed a numpy array with inconsistent
```

```
    dimensions). The dynamic calculation will then be perform th
anks to
    'seqlen' attribute that records every actual sequence length
.
    """
    def __init__(self, n_samples=1000, max_seq_len=20, min_seq_l
en=3,
                 max_value=1000):
        self.data = []
        self.labels = []
        self.seqlen = []
        for i in range(n_samples):
            # Random sequence length
            len = random.randint(min_seq_len, max_seq_len)
            # Monitor sequence length for TensorFlow dynamic cal
culation
            self.seqlen.append(len)
            # Add a random or linear int sequence (50% prob)
            if random.random() < .5:
                # Generate a linear sequence
                rand_start = random.randint(0, max_value - len)
                s = [[float(i)/max_value] for i in
                     range(rand_start, rand_start + len)]
                # Pad sequence for dimension consistency
                s += [[0.] for i in range(max_seq_len - len)]
                self.data.append(s)
                self.labels.append([1., 0.])
            else:
                # Generate a random sequence
                s = [[float(random.randint(0, max_value))/max_va
lue]
                     for i in range(len)]
                # Pad sequence for dimension consistency
                s += [[0.] for i in range(max_seq_len - len)]
                self.data.append(s)
                self.labels.append([0., 1.])
        self.batch_id = 0

    def next(self, batch_size):
        """ Return a batch of data. When dataset end is reached,
 start over.
        """
        if self.batch_id == len(self.data):
            self.batch_id = 0
        batch_data = (self.data[self.batch_id:min(self.batch_id
+
                                                  batch_size, le
n(self.data))])
        batch_labels = (self.labels[self.batch_id:min(self.batch
_id +
                                                  batch_size, le
n(self.data))])
        batch_seqlen = (self.seqlen[self.batch_id:min(self.batch
```

```
_id +
                                                  batch_size, le
n(self.data))])
        self.batch_id = min(self.batch_id + batch_size, len(self
.data))
        return batch_data, batch_labels, batch_seqlen
```

```
# ==========
#   MODEL
# ==========

# Parameters
learning_rate = 0.01
training_steps = 10000
batch_size = 128
display_step = 200

# Network Parameters
seq_max_len = 20 # Sequence max length
n_hidden = 64 # hidden layer num of features
n_classes = 2 # linear sequence or not

trainset = ToySequenceData(n_samples=1000, max_seq_len=seq_max_l
en)
testset = ToySequenceData(n_samples=500, max_seq_len=seq_max_len
)

# tf Graph input
x = tf.placeholder("float", [None, seq_max_len, 1])
y = tf.placeholder("float", [None, n_classes])
# A placeholder for indicating each sequence length
seqlen = tf.placeholder(tf.int32, [None])

# Define weights
weights = {
    'out': tf.Variable(tf.random_normal([n_hidden, n_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

```
def dynamicRNN(x, seqlen, weights, biases):

    # Prepare data shape to match `rnn` function requirements
    # Current data input shape: (batch_size, n_steps, n_input)
    # Required shape: 'n_steps' tensors list of shape (batch_siz
e, n_input)

    # Unstack to get a list of 'n_steps' tensors of shape (batch
_size, n_input)
    x = tf.unstack(x, seq_max_len, 1)

    # Define a lstm cell with tensorflow
    lstm_cell = tf.contrib.rnn.BasicLSTMCell(n_hidden)

    # Get lstm cell output, providing 'sequence_length' will per
form dynamic
    # calculation.
    outputs, states = tf.contrib.rnn.static_rnn(lstm_cell, x, dt
ype=tf.float32,
                                sequence_length=seqlen)

    # When performing dynamic calculation, we must retrieve the
last
    # dynamically computed output, i.e., if a sequence length is
 10, we need
    # to retrieve the 10th output.
    # However TensorFlow doesn't support advanced indexing yet,
so we build
    # a custom op that for each sample in batch size, get its le
ngth and
    # get the corresponding relevant output.

    # 'outputs' is a list of output at every timestep, we pack t
hem in a Tensor
    # and change back dimension to [batch_size, n_step, n_input]
    outputs = tf.stack(outputs)
    outputs = tf.transpose(outputs, [1, 0, 2])

    # Hack to build the indexing and retrieve the right output.
    batch_size = tf.shape(outputs)[0]
    # Start indices for each sample
    index = tf.range(0, batch_size) * seq_max_len + (seqlen - 1)
    # Indexing
    outputs = tf.gather(tf.reshape(outputs, [-1, n_hidden]), ind
ex)

    # Linear activation, using outputs computed above
    return tf.matmul(outputs, weights['out']) + biases['out']
```

```
pred = dynamicRNN(x, seqlen, weights, biases)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=pred, labels=y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=lear
ning_rate).minimize(cost)

# Evaluate model
correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
/Users/aymeric.damien/anaconda2/lib/python2.7/site-packages/tens
orflow/python/ops/gradients_impl.py:93: UserWarning: Converting
sparse IndexedSlices to a dense Tensor of unknown shape. This ma
y consume a large amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown
shape. "
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, training_steps+1):
        batch_x, batch_y, batch_seqlen = trainset.next(batch_siz
e)
        # Run optimization op (backprop)
        sess.run(optimizer, feed_dict={x: batch_x, y: batch_y,
                                       seqlen: batch_seqlen})
        if step % display_step == 0 or step == 1:
            # Calculate batch accuracy & loss
            acc, loss = sess.run([accuracy, cost], feed_dict={x:
 batch_x, y: batch_y,
                                                    seqlen: batch_se
qlen})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.6f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.5f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy
    test_data = testset.data
    test_label = testset.labels
    test_seqlen = testset.seqlen
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={x: test_data, y: test_labe
l,
                                      seqlen: test_seqlen}))
```

```
Step 1, Minibatch Loss= 0.864517, Training Accuracy= 0.42188
Step 200, Minibatch Loss= 0.686012, Training Accuracy= 0.43269
Step 400, Minibatch Loss= 0.682970, Training Accuracy= 0.48077
Step 600, Minibatch Loss= 0.679640, Training Accuracy= 0.50962
Step 800, Minibatch Loss= 0.675208, Training Accuracy= 0.53846
Step 1000, Minibatch Loss= 0.668636, Training Accuracy= 0.56731
Step 1200, Minibatch Loss= 0.657525, Training Accuracy= 0.62500
Step 1400, Minibatch Loss= 0.635423, Training Accuracy= 0.67308
Step 1600, Minibatch Loss= 0.580433, Training Accuracy= 0.75962
Step 1800, Minibatch Loss= 0.475599, Training Accuracy= 0.81731
Step 2000, Minibatch Loss= 0.434865, Training Accuracy= 0.83654
Step 2200, Minibatch Loss= 0.423690, Training Accuracy= 0.85577
Step 2400, Minibatch Loss= 0.417472, Training Accuracy= 0.85577
Step 2600, Minibatch Loss= 0.412906, Training Accuracy= 0.85577
Step 2800, Minibatch Loss= 0.409193, Training Accuracy= 0.85577
Step 3000, Minibatch Loss= 0.406035, Training Accuracy= 0.86538
```

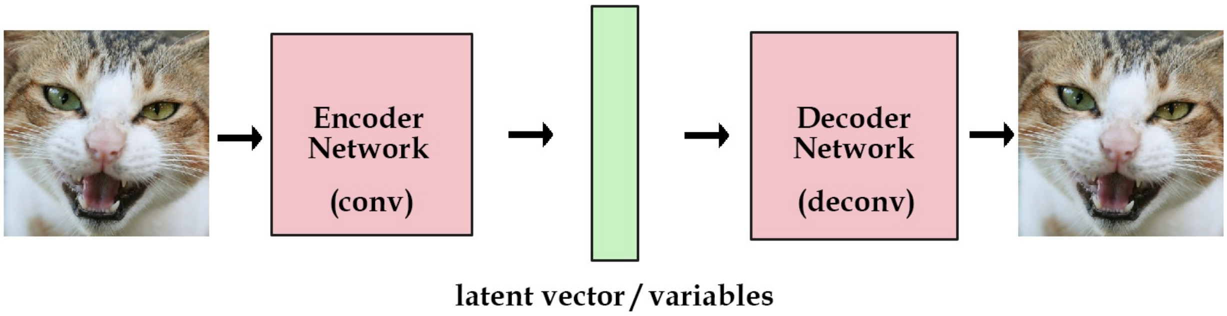
```
Step 3200, Minibatch Loss= 0.403287, Training Accuracy= 0.87500
Step 3400, Minibatch Loss= 0.400862, Training Accuracy= 0.87500
Step 3600, Minibatch Loss= 0.398704, Training Accuracy= 0.86538
Step 3800, Minibatch Loss= 0.396768, Training Accuracy= 0.86538
Step 4000, Minibatch Loss= 0.395017, Training Accuracy= 0.86538
Step 4200, Minibatch Loss= 0.393422, Training Accuracy= 0.86538
Step 4400, Minibatch Loss= 0.391957, Training Accuracy= 0.85577
Step 4600, Minibatch Loss= 0.390600, Training Accuracy= 0.85577
Step 4800, Minibatch Loss= 0.389334, Training Accuracy= 0.86538
Step 5000, Minibatch Loss= 0.388143, Training Accuracy= 0.86538
Step 5200, Minibatch Loss= 0.387015, Training Accuracy= 0.86538
Step 5400, Minibatch Loss= 0.385940, Training Accuracy= 0.86538
Step 5600, Minibatch Loss= 0.384907, Training Accuracy= 0.86538
Step 5800, Minibatch Loss= 0.383904, Training Accuracy= 0.85577
Step 6000, Minibatch Loss= 0.382921, Training Accuracy= 0.86538
Step 6200, Minibatch Loss= 0.381941, Training Accuracy= 0.86538
Step 6400, Minibatch Loss= 0.380947, Training Accuracy= 0.86538
Step 6600, Minibatch Loss= 0.379912, Training Accuracy= 0.86538
Step 6800, Minibatch Loss= 0.378796, Training Accuracy= 0.86538
Step 7000, Minibatch Loss= 0.377540, Training Accuracy= 0.86538
Step 7200, Minibatch Loss= 0.376041, Training Accuracy= 0.86538
Step 7400, Minibatch Loss= 0.374130, Training Accuracy= 0.85577
Step 7600, Minibatch Loss= 0.371514, Training Accuracy= 0.85577
Step 7800, Minibatch Loss= 0.367723, Training Accuracy= 0.85577
Step 8000, Minibatch Loss= 0.362049, Training Accuracy= 0.85577
Step 8200, Minibatch Loss= 0.353558, Training Accuracy= 0.85577
Step 8400, Minibatch Loss= 0.341072, Training Accuracy= 0.86538
Step 8600, Minibatch Loss= 0.323062, Training Accuracy= 0.87500
Step 8800, Minibatch Loss= 0.299278, Training Accuracy= 0.89423
Step 9000, Minibatch Loss= 0.273857, Training Accuracy= 0.90385
Step 9200, Minibatch Loss= 0.248392, Training Accuracy= 0.91346
Step 9400, Minibatch Loss= 0.221348, Training Accuracy= 0.92308
Step 9600, Minibatch Loss= 0.191947, Training Accuracy= 0.92308
Step 9800, Minibatch Loss= 0.159308, Training Accuracy= 0.93269
Step 10000, Minibatch Loss= 0.136938, Training Accuracy= 0.96154
Optimization Finished!
Testing Accuracy: 0.952
```

Auto-Encoder Example

Build a 2 layers auto-encoder with TensorFlow to compress images to a lower latent space and then reconstruct them.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

Auto-Encoder Overview



References:

- Gradient-based learning applied to document recognition. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Proceedings of the IEEE, 86(11):2278-2324, November 1998.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Parameters
learning_rate = 0.01
num_steps = 30000
batch_size = 256

display_step = 1000
examples_to_show = 10

# Network Parameters
num_hidden_1 = 256 # 1st layer num features
num_hidden_2 = 128 # 2nd layer num features (the latent dim)
num_input = 784 # MNIST data input (img shape: 28*28)

# tf Graph input (only pictures)
X = tf.placeholder("float", [None, num_input])

weights = {
    'encoder_h1': tf.Variable(tf.random_normal([num_input, num_h
idden_1])),
    'encoder_h2': tf.Variable(tf.random_normal([num_hidden_1, nu
m_hidden_2])),
    'decoder_h1': tf.Variable(tf.random_normal([num_hidden_2, nu
m_hidden_1])),
    'decoder_h2': tf.Variable(tf.random_normal([num_hidden_1, nu
m_input])),
}
biases = {
    'encoder_b1': tf.Variable(tf.random_normal([num_hidden_1])),
    'encoder_b2': tf.Variable(tf.random_normal([num_hidden_2])),
    'decoder_b1': tf.Variable(tf.random_normal([num_hidden_1])),
    'decoder_b2': tf.Variable(tf.random_normal([num_input])),
}
```

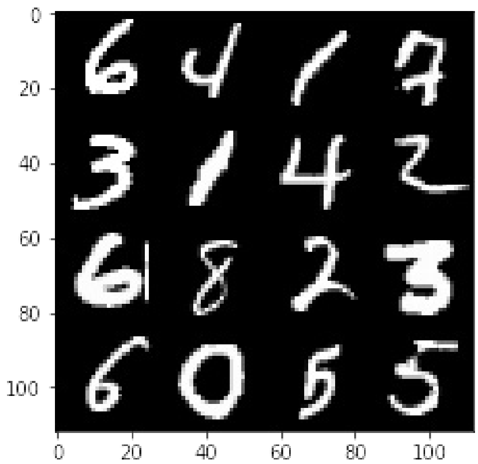
```
# Building the encoder
def encoder(x):
    # Encoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['encoder_h1']),
                                   biases['encoder_b1']))
    # Encoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['encoder_h2']),
                                   biases['encoder_b2']))
    return layer_2


# Building the decoder
def decoder(x):
    # Decoder Hidden layer with sigmoid activation #1
    layer_1 = tf.nn.sigmoid(tf.add(tf.matmul(x, weights['decoder_h1']),
                                   biases['decoder_b1']))
    # Decoder Hidden layer with sigmoid activation #2
    layer_2 = tf.nn.sigmoid(tf.add(tf.matmul(layer_1, weights['decoder_h2']),
                                   biases['decoder_b2']))
    return layer_2

# Construct model
encoder_op = encoder(X)
decoder_op = decoder(encoder_op)

# Prediction
y_pred = decoder_op
# Targets (Labels) are the input data.
y_true = X

# Define loss and optimizer, minimize the squared error
loss = tf.reduce_mean(tf.pow(y_true - y_pred, 2))
optimizer = tf.train.RMSPropOptimizer(learning_rate).minimize(loss)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start Training
# Start a new TF session
sess = tf.Session()

# Run the initializer
sess.run(init)

# Training
for i in range(1, num_steps+1):
    # Prepare Data
    # Get the next batch of MNIST data (only images are needed,
not labels)
    batch_x, _ = mnist.train.next_batch(batch_size)

    # Run optimization op (backprop) and cost op (to get loss va
lue)
    _, l = sess.run([optimizer, loss], feed_dict={X: batch_x})
    # Display logs per step
    if i % display_step == 0 or i == 1:
        print('Step %i: Minibatch Loss: %f' % (i, l))
```

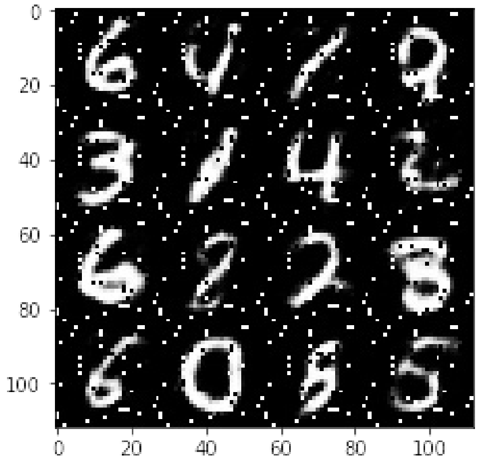
```
Step 1: Minibatch Loss: 0.438300
Step 1000: Minibatch Loss: 0.146586
Step 2000: Minibatch Loss: 0.130722
Step 3000: Minibatch Loss: 0.117178
Step 4000: Minibatch Loss: 0.109027
Step 5000: Minibatch Loss: 0.102582
Step 6000: Minibatch Loss: 0.099183
Step 7000: Minibatch Loss: 0.095619
Step 8000: Minibatch Loss: 0.089006
Step 9000: Minibatch Loss: 0.087125
Step 10000: Minibatch Loss: 0.083930
Step 11000: Minibatch Loss: 0.077512
Step 12000: Minibatch Loss: 0.077137
Step 13000: Minibatch Loss: 0.073983
Step 14000: Minibatch Loss: 0.074218
Step 15000: Minibatch Loss: 0.074492
Step 16000: Minibatch Loss: 0.074374
Step 17000: Minibatch Loss: 0.070909
Step 18000: Minibatch Loss: 0.069438
Step 19000: Minibatch Loss: 0.068245
Step 20000: Minibatch Loss: 0.068402
Step 21000: Minibatch Loss: 0.067113
Step 22000: Minibatch Loss: 0.068241
Step 23000: Minibatch Loss: 0.062454
Step 24000: Minibatch Loss: 0.059754
Step 25000: Minibatch Loss: 0.058687
Step 26000: Minibatch Loss: 0.059107
Step 27000: Minibatch Loss: 0.055788
Step 28000: Minibatch Loss: 0.057263
Step 29000: Minibatch Loss: 0.056391
Step 30000: Minibatch Loss: 0.057672
```

```
# Testing
# Encode and decode images from test set and visualize their rec
onstruction.
n = 4
canvas_orig = np.empty((28 * n, 28 * n))
canvas_recon = np.empty((28 * n, 28 * n))
for i in range(n):
    # MNIST test set
    batch_x, _ = mnist.test.next_batch(n)
    # Encode and decode the digit image
    g = sess.run(decoder_op, feed_dict={X: batch_x})

    # Display original images
    for j in range(n):
        # Draw the generated digits
        canvas_orig[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] =
batch_x[j].reshape([28, 28])
    # Display reconstructed images
    for j in range(n):
        # Draw the generated digits
        canvas_recon[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] =
 g[j].reshape([28, 28])

print("Original Images")
plt.figure(figsize=(n, n))
plt.imshow(canvas_orig, origin="upper", cmap="gray")
plt.show()

print("Reconstructed Images")
plt.figure(figsize=(n, n))
plt.imshow(canvas_recon, origin="upper", cmap="gray")
plt.show()
```

```
Original Images
```



```
Reconstructed Images
```

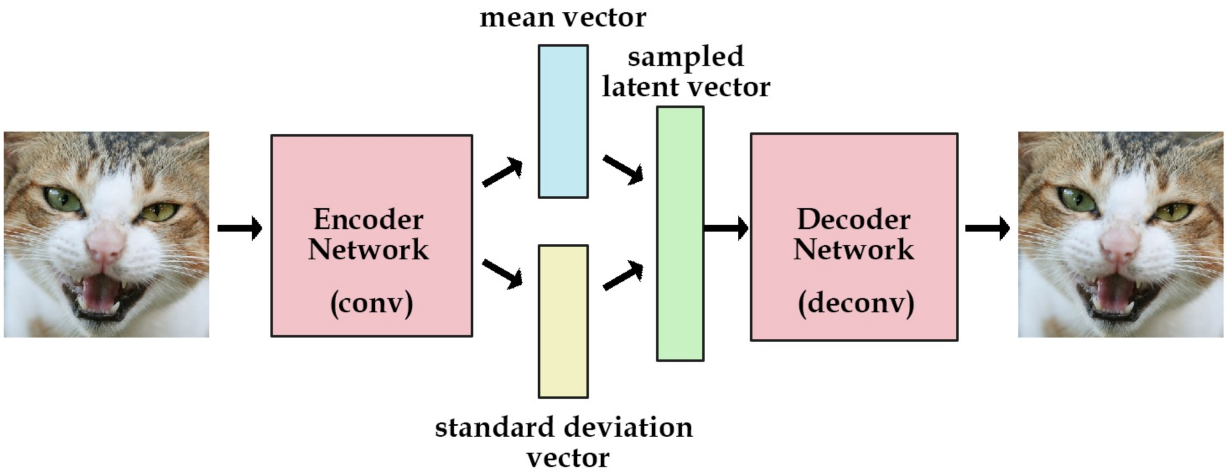


Variational Auto-Encoder Example

Build a variational auto-encoder (VAE) to generate digit images from a noise distribution with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

VAE Overview



References:

- Auto-Encoding Variational Bayes The International Conference on Learning Representations (ICLR), Banff, 2014. D.P. Kingma, M. Welling
- Understanding the difficulty of training deep feedforward neural networks. X Glorot, Y Bengio. Aistats 9, 249-256

Other tutorials:

- Variational Auto Encoder Explained. Kevin Frans.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import tensorflow as tf
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes
.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.001
num_steps = 30000
batch_size = 64

# Network Parameters
image_dim = 784 # MNIST images are 28x28 pixels
hidden_dim = 512
latent_dim = 2

# A custom initialization (see Xavier Glorot init)
def glorot_init(shape):
    return tf.random_normal(shape=shape, stddev=1. / tf.sqrt(sha
pe[0] / 2.))
```

```
# Variables
weights = {
    'encoder_h1': tf.Variable(glorot_init([image_dim, hidden_dim
])),
    'z_mean': tf.Variable(glorot_init([hidden_dim, latent_dim]))
,
    'z_std': tf.Variable(glorot_init([hidden_dim, latent_dim])),
    'decoder_h1': tf.Variable(glorot_init([latent_dim, hidden_di
m])),
    'decoder_out': tf.Variable(glorot_init([hidden_dim, image_di
m]))
}
biases = {
    'encoder_b1': tf.Variable(glorot_init([hidden_dim])),
    'z_mean': tf.Variable(glorot_init([latent_dim])),
    'z_std': tf.Variable(glorot_init([latent_dim])),
    'decoder_b1': tf.Variable(glorot_init([hidden_dim])),
    'decoder_out': tf.Variable(glorot_init([image_dim]))
}
```

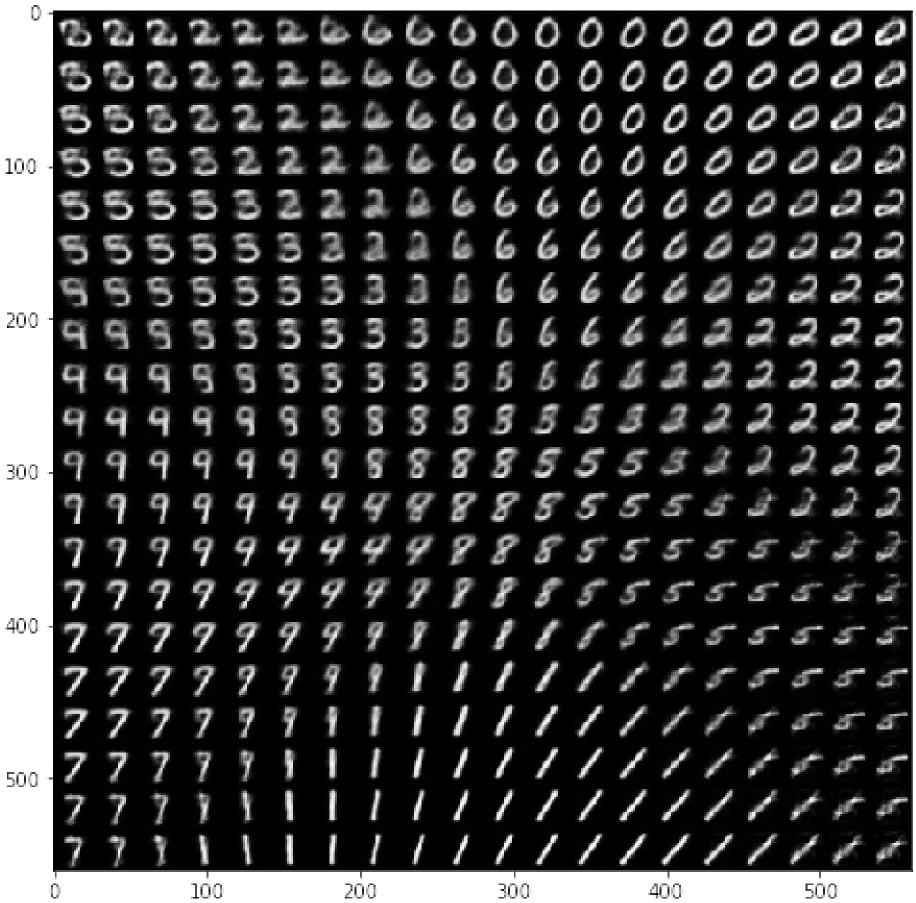
```
# Building the encoder
input_image = tf.placeholder(tf.float32, shape=[None, image_dim]
)
encoder = tf.matmul(input_image, weights['encoder_h1']) + biases[
'encoder_b1']
encoder = tf.nn.tanh(encoder)
z_mean = tf.matmul(encoder, weights['z_mean']) + biases['z_mean'
]
z_std = tf.matmul(encoder, weights['z_std']) + biases['z_std']

# Sampler: Normal (gaussian) random distribution
eps = tf.random_normal(tf.shape(z_std), dtype=tf.float32, mean=0.
, stddev=1.0,
                       name='epsilon')
z = z_mean + tf.exp(z_std / 2) * eps

# Building the decoder (with scope to re-use these layers later)
decoder = tf.matmul(z, weights['decoder_h1']) + biases['decoder_
b1']
decoder = tf.nn.tanh(decoder)
decoder = tf.matmul(decoder, weights['decoder_out']) + biases['d
ecoder_out']
decoder = tf.nn.sigmoid(decoder)
```

```
# Define VAE Loss
def vae_loss(x_reconstructed, x_true):
    # Reconstruction loss
    encode_decode_loss = x_true * tf.log(1e-10 + x_reconstructed
) \
                         + (1 - x_true) * tf.log(1e-10 + 1 - x_r
econstructed)
    encode_decode_loss = -tf.reduce_sum(encode_decode_loss, 1)
    # KL Divergence loss
    kl_div_loss = 1 + z_std - tf.square(z_mean) - tf.exp(z_std)
    kl_div_loss = -0.5 * tf.reduce_sum(kl_div_loss, 1)
    return tf.reduce_mean(encode_decode_loss + kl_div_loss)

loss_op = vae_loss(decoder, input_image)
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rat
e)
train_op = optimizer.minimize(loss_op)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start Training
# Start a new TF session
sess = tf.Session()

# Run the initializer
sess.run(init)

# Training
for i in range(1, num_steps+1):
    # Prepare Data
    # Get the next batch of MNIST data (only images are needed,
not labels)
    batch_x, _ = mnist.train.next_batch(batch_size)

    # Train
    feed_dict = {input_image: batch_x}
    _, l = sess.run([train_op, loss_op], feed_dict=feed_dict)
    if i % 1000 == 0 or i == 1:
        print('Step %i, Loss: %f' % (i, l))
```

```
Step 1, Loss: 645.076538
Step 1000, Loss: 173.018188
Step 2000, Loss: 165.299225
Step 3000, Loss: 172.933685
Step 4000, Loss: 161.475052
Step 5000, Loss: 179.529831
Step 6000, Loss: 166.430023
Step 7000, Loss: 167.152176
Step 8000, Loss: 159.920242
Step 9000, Loss: 160.172363
Step 10000, Loss: 150.077652
Step 11000, Loss: 162.774567
Step 12000, Loss: 156.187820
Step 13000, Loss: 148.331573
Step 14000, Loss: 153.757202
Step 15000, Loss: 158.050598
Step 16000, Loss: 163.068939
Step 17000, Loss: 152.765152
Step 18000, Loss: 151.136353
Step 19000, Loss: 157.889664
Step 20000, Loss: 149.112473
Step 21000, Loss: 151.694885
Step 22000, Loss: 153.153229
Step 23000, Loss: 152.662323
Step 24000, Loss: 150.556198
Step 25000, Loss: 142.779984
Step 26000, Loss: 148.985382
Step 27000, Loss: 150.923401
Step 28000, Loss: 161.761551
Step 29000, Loss: 144.045578
Step 30000, Loss: 151.272964
```

```
# Testing
# Generator takes noise as input
noise_input = tf.placeholder(tf.float32, shape=[None, latent_dim
])
# Rebuild the decoder to create image from noise
decoder = tf.matmul(noise_input, weights['decoder_h1']) + biases[
'decoder_b1']
decoder = tf.nn.tanh(decoder)
decoder = tf.matmul(decoder, weights['decoder_out']) + biases['d
ecoder_out']
decoder = tf.nn.sigmoid(decoder)

# Building a manifold of generated digits
n = 20
x_axis = np.linspace(-3, 3, n)
y_axis = np.linspace(-3, 3, n)

canvas = np.empty((28 * n, 28 * n))
for i, yi in enumerate(x_axis):
    for j, xi in enumerate(y_axis):
        z_mu = np.array([[xi, yi]] * batch_size)
        x_mean = sess.run(decoder, feed_dict={noise_input: z_mu}
)
        canvas[(n - i - 1) * 28:(n - i) * 28, j * 28:(j + 1) * 28
] = \
        x_mean[0].reshape(28, 28)

plt.figure(figsize=(8, 10))
Xi, Yi = np.meshgrid(x_axis, y_axis)
plt.imshow(canvas, origin="upper", cmap="gray")
plt.show()
```

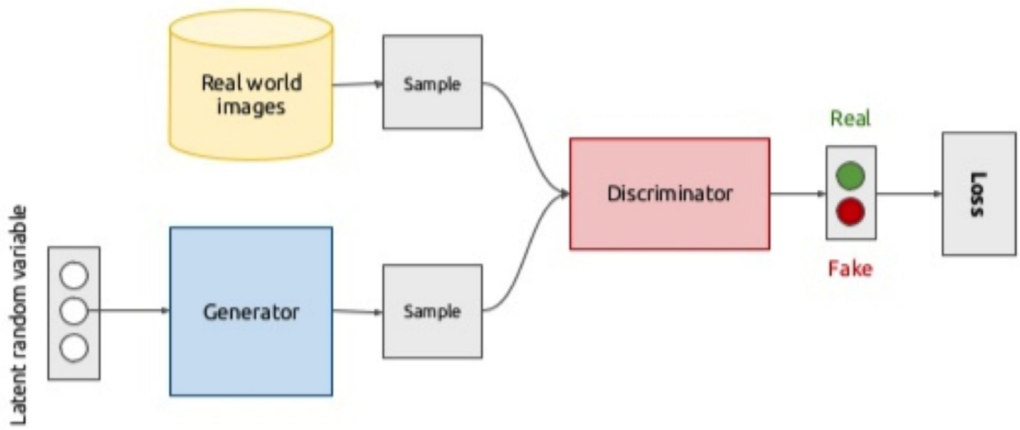

0
100
200
300
400
500
0
100
200
300
400
500

Generative Adversarial Network Example

Build a generative adversarial network (GAN) to generate digit images from a noise distribution with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

GAN Overview



References:

- Generative adversarial nets. I Goodfellow, J Pouget-Abadie, M Mirza, B Xu, D Warde-Farley, S Ozair, Y. Bengio. Advances in neural information processing systems, 2672-2680.
- Understanding the difficulty of training deep feedforward neural networks. X Glorot, Y Bengio. Aistats 9, 249-256

Other tutorials:

- Generative Adversarial Networks Explained. Kevin Frans.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Params
num_steps = 70000
batch_size = 128
learning_rate = 0.0002

# Network Params
image_dim = 784 # 28*28 pixels
gen_hidden_dim = 256
disc_hidden_dim = 256
noise_dim = 100 # Noise data points

# A custom initialization (see Xavier Glorot init)
def glorot_init(shape):
    return tf.random_normal(shape=shape, stddev=1. / tf.sqrt(sha
pe[0] / 2.))
```

```
# Store layers weight & bias
weights = {
    'gen_hidden1': tf.Variable(glorot_init([noise_dim, gen_hidde
n_dim])),
    'gen_out': tf.Variable(glorot_init([gen_hidden_dim, image_di
m])),
    'disc_hidden1': tf.Variable(glorot_init([image_dim, disc_hid
den_dim])),
    'disc_out': tf.Variable(glorot_init([disc_hidden_dim, 1])),
}
biases = {
    'gen_hidden1': tf.Variable(tf.zeros([gen_hidden_dim])),
    'gen_out': tf.Variable(tf.zeros([image_dim])),
    'disc_hidden1': tf.Variable(tf.zeros([disc_hidden_dim])),
    'disc_out': tf.Variable(tf.zeros([1])),
}
```

```
# Generator
def generator(x):
    hidden_layer = tf.matmul(x, weights['gen_hidden1'])
    hidden_layer = tf.add(hidden_layer, biases['gen_hidden1'])
    hidden_layer = tf.nn.relu(hidden_layer)
    out_layer = tf.matmul(hidden_layer, weights['gen_out'])
    out_layer = tf.add(out_layer, biases['gen_out'])
    out_layer = tf.nn.sigmoid(out_layer)
    return out_layer


# Discriminator
def discriminator(x):
    hidden_layer = tf.matmul(x, weights['disc_hidden1'])
    hidden_layer = tf.add(hidden_layer, biases['disc_hidden1'])
    hidden_layer = tf.nn.relu(hidden_layer)
    out_layer = tf.matmul(hidden_layer, weights['disc_out'])
    out_layer = tf.add(out_layer, biases['disc_out'])
    out_layer = tf.nn.sigmoid(out_layer)
    return out_layer

# Build Networks
# Network Inputs
gen_input = tf.placeholder(tf.float32, shape=[None, noise_dim],
name='input_noise')
disc_input = tf.placeholder(tf.float32, shape=[None, image_dim],
 name='disc_input')

# Build Generator Network
gen_sample = generator(gen_input)

# Build 2 Discriminator Networks (one from noise input, one from
 generated samples)
```

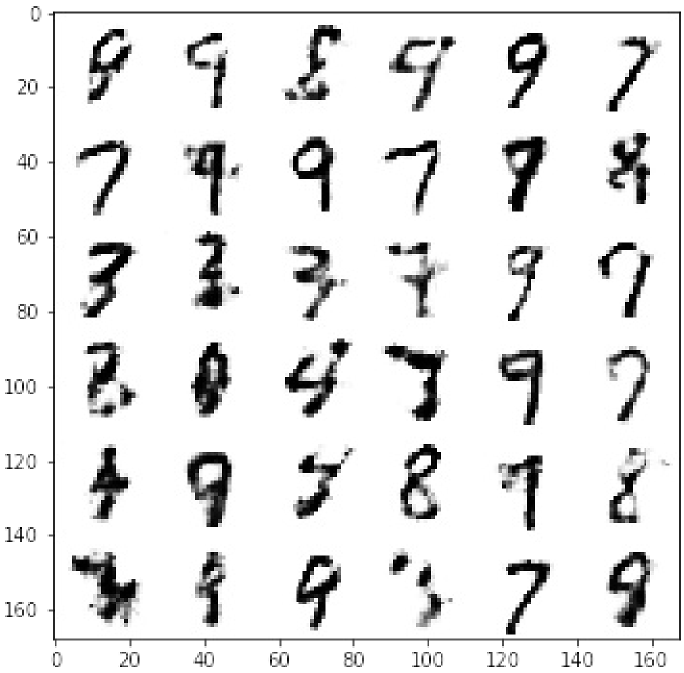
```
disc_real = discriminator(disc_input)
disc_fake = discriminator(gen_sample)

# Build Loss
gen_loss = -tf.reduce_mean(tf.log(disc_fake))
disc_loss = -tf.reduce_mean(tf.log(disc_real) + tf.log(1. - disc
_fake))

# Build Optimizers
optimizer_gen = tf.train.AdamOptimizer(learning_rate=learning_ra
te)
optimizer_disc = tf.train.AdamOptimizer(learning_rate=learning_r
ate)

# Training Variables for each optimizer
# By default in TensorFlow, all variables are updated by each op
timizer, so we
# need to precise for each one of them the specific variables to
 update.
# Generator Network Variables
gen_vars = [weights['gen_hidden1'], weights['gen_out'],
            biases['gen_hidden1'], biases['gen_out']]
# Discriminator Network Variables
disc_vars = [weights['disc_hidden1'], weights['disc_out'],
            biases['disc_hidden1'], biases['disc_out']]

# Create training operations
train_gen = optimizer_gen.minimize(gen_loss, var_list=gen_vars)
train_disc = optimizer_disc.minimize(disc_loss, var_list=disc_va
rs)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start Training
# Start a new TF session
sess = tf.Session()

# Run the initializer
sess.run(init)

# Training
for i in range(1, num_steps+1):
    # Prepare Data
    # Get the next batch of MNIST data (only images are needed, not labels)
    batch_x, _ = mnist.train.next_batch(batch_size)
    # Generate noise to feed to the generator
    z = np.random.uniform(-1., 1., size=[batch_size, noise_dim])

    # Train
    feed_dict = {disc_input: batch_x, gen_input: z}
    _, _, gl, dl = sess.run([train_gen, train_disc, gen_loss, disc_loss],
                            feed_dict=feed_dict)
    if i % 2000 == 0 or i == 1:
        print('Step %i: Generator Loss: %f, Discriminator Loss: %f' % (i, gl, dl))
```

```
Step 1: Generator Loss: 0.774581, Discriminator Loss: 1.300602
Step 2000: Generator Loss: 4.521158, Discriminator Loss: 0.030166
Step 4000: Generator Loss: 3.685439, Discriminator Loss: 0.125958
Step 6000: Generator Loss: 4.412449, Discriminator Loss: 0.097088
Step 8000: Generator Loss: 3.996747, Discriminator Loss: 0.150800
Step 10000: Generator Loss: 3.850827, Discriminator Loss: 0.225699
Step 12000: Generator Loss: 2.950704, Discriminator Loss: 0.279967
Step 14000: Generator Loss: 3.741951, Discriminator Loss: 0.241062
Step 16000: Generator Loss: 3.117743, Discriminator Loss: 0.432293
Step 18000: Generator Loss: 3.647199, Discriminator Loss: 0.278121
Step 20000: Generator Loss: 3.186711, Discriminator Loss: 0.313830
Step 22000: Generator Loss: 3.737114, Discriminator Loss: 0.201730
Step 24000: Generator Loss: 3.042442, Discriminator Loss: 0.454414
```

```
Step 26000: Generator Loss: 3.340376, Discriminator Loss: 0.2494
28
Step 28000: Generator Loss: 3.423218, Discriminator Loss: 0.3696
53
Step 30000: Generator Loss: 3.219242, Discriminator Loss: 0.4635
35
Step 32000: Generator Loss: 3.313017, Discriminator Loss: 0.2760
70
Step 34000: Generator Loss: 3.413397, Discriminator Loss: 0.3677
21
Step 36000: Generator Loss: 3.240625, Discriminator Loss: 0.4461
60
Step 38000: Generator Loss: 3.175355, Discriminator Loss: 0.3776
28
Step 40000: Generator Loss: 3.154558, Discriminator Loss: 0.4788
12
Step 42000: Generator Loss: 3.210753, Discriminator Loss: 0.4975
02
Step 44000: Generator Loss: 2.883431, Discriminator Loss: 0.3958
12
Step 46000: Generator Loss: 2.584176, Discriminator Loss: 0.4207
83
Step 48000: Generator Loss: 2.581381, Discriminator Loss: 0.4692
89
Step 50000: Generator Loss: 2.752729, Discriminator Loss: 0.3735
44
Step 52000: Generator Loss: 2.649749, Discriminator Loss: 0.4637
55
Step 54000: Generator Loss: 2.468188, Discriminator Loss: 0.5561
29
Step 56000: Generator Loss: 2.653330, Discriminator Loss: 0.3775
72
Step 58000: Generator Loss: 2.697943, Discriminator Loss: 0.4241
33
Step 60000: Generator Loss: 2.835973, Discriminator Loss: 0.4132
52
Step 62000: Generator Loss: 2.751346, Discriminator Loss: 0.4033
32
Step 64000: Generator Loss: 3.212001, Discriminator Loss: 0.5344
27
Step 66000: Generator Loss: 2.878227, Discriminator Loss: 0.4312
44
Step 68000: Generator Loss: 3.104266, Discriminator Loss: 0.4268
25
Step 70000: Generator Loss: 2.871485, Discriminator Loss: 0.3486
38
```

```
# Testing
# Generate images from noise, using the generator network.
n = 6
canvas = np.empty((28 * n, 28 * n))
for i in range(n):
    # Noise input.
    z = np.random.uniform(-1., 1., size=[n, noise_dim])
    # Generate image from noise.
    g = sess.run(gen_sample, feed_dict={gen_input: z})
    # Reverse colours for better display
    g = -1 * (g - 1)
    for j in range(n):
        # Draw the generated digits
        canvas[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] = g[j].
reshape([28, 28])

plt.figure(figsize=(n, n))
plt.imshow(canvas, origin="upper", cmap="gray")
plt.show()
```

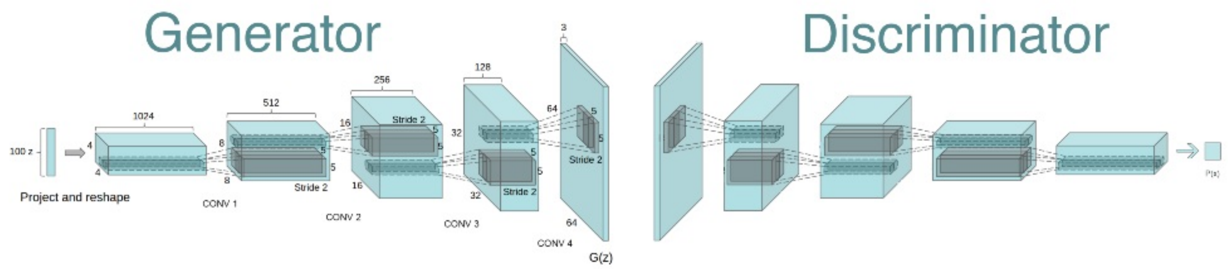


Deep Convolutional Generative Adversarial Network Example

Build a deep convolutional generative adversarial network (DCGAN) to generate digit images from a noise distribution with TensorFlow.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

DCGAN Overview



References:

- Unsupervised representation learning with deep convolutional generative adversarial networks. A Radford, L Metz, S Chintala, 2016.
- Understanding the difficulty of training deep feedforward neural networks. X Glorot, Y Bengio. Aistats 9, 249-256
- Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Sergey Ioffe, Christian Szegedy. 2015.

MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flattened and converted to a 1-D numpy array of 784 features (28*28).

More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import division, print_function, absolute_import

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
```

```
# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Training Params
num_steps = 10000
batch_size = 128
lr_generator = 0.002
lr_discriminator = 0.002

# Network Params
image_dim = 784 # 28*28 pixels * 1 channel
noise_dim = 100 # Noise data points
```

```
# Build Networks
# Network Inputs
noise_input = tf.placeholder(tf.float32, shape=[None, noise_dim]
)
real_image_input = tf.placeholder(tf.float32, shape=[None, 28, 28
, 1])
# A boolean to indicate batch normalization if it is training or
```

```
 inference time
is_training = tf.placeholder(tf.bool)

#LeakyReLU activation
def leakyrelu(x, alpha=0.2):
    return 0.5 * (1 + alpha) * x + 0.5 * (1 - alpha) * abs(x)

# Generator Network
# Input: Noise, Output: Image
# Note that batch normalization has different behavior at traini
ng and inference time,
# we then use a placeholder to indicates the layer if we are tra
ining or not.
def generator(x, reuse=False):
    with tf.variable_scope('Generator', reuse=reuse):
        # TensorFlow Layers automatically create variables and c
alculate their
        # shape, based on the input.
        x = tf.layers.dense(x, units=7 * 7 * 128)
        x = tf.layers.batch_normalization(x, training=is_trainin
g)
        x = tf.nn.relu(x)
        # Reshape to a 4-D array of images: (batch, height, widt
h, channels)
        # New shape: (batch, 7, 7, 128)
        x = tf.reshape(x, shape=[-1, 7, 7, 128])
        # Deconvolution, image shape: (batch, 14, 14, 64)
        x = tf.layers.conv2d_transpose(x, 64, 5, strides=2, padd
ing='same')
        x = tf.layers.batch_normalization(x, training=is_trainin
g)
        x = tf.nn.relu(x)
        # Deconvolution, image shape: (batch, 28, 28, 1)
        x = tf.layers.conv2d_transpose(x, 1, 5, strides=2, paddi
ng='same')
        # Apply tanh for better stability - clip values to [-1,
1].
        x = tf.nn.tanh(x)
        return x


# Discriminator Network
# Input: Image, Output: Prediction Real/Fake Image
def discriminator(x, reuse=False):
    with tf.variable_scope('Discriminator', reuse=reuse):
        # Typical convolutional neural network to classify image
s.
        x = tf.layers.conv2d(x, 64, 5, strides=2, padding='same'
)
        x = tf.layers.batch_normalization(x, training=is_trainin
g)
        x = leakyrelu(x)
        x = tf.layers.conv2d(x, 128, 5, strides=2, padding='same'
```

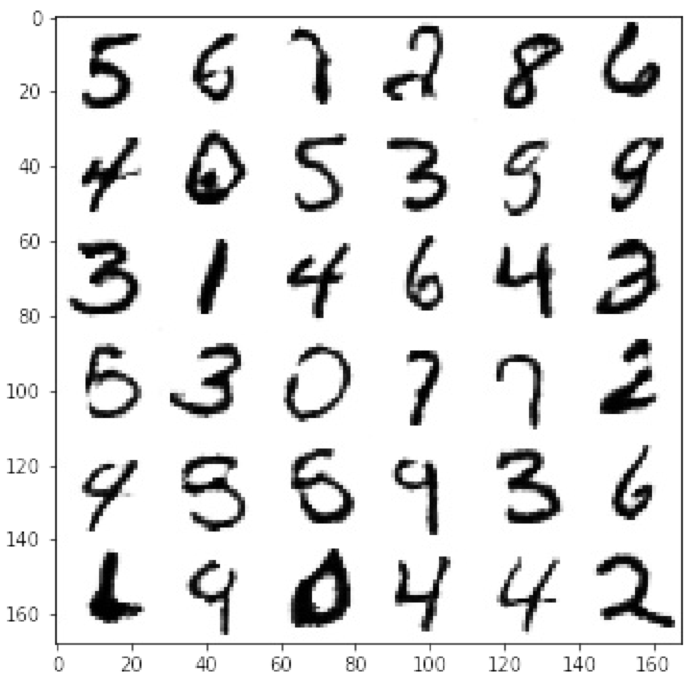
```
)
        x = tf.layers.batch_normalization(x, training=is_trainin
g)
        x = leakyrelu(x)
        # Flatten
        x = tf.reshape(x, shape=[-1, 7*7*128])
        x = tf.layers.dense(x, 1024)
        x = tf.layers.batch_normalization(x, training=is_trainin
g)
        x = leakyrelu(x)
        # Output 2 classes: Real and Fake images
        x = tf.layers.dense(x, 2)
    return x

# Build Generator Network
gen_sample = generator(noise_input)

# Build 2 Discriminator Networks (one from noise input, one from
 generated samples)
disc_real = discriminator(real_image_input)
disc_fake = discriminator(gen_sample, reuse=True)

# Build the stacked generator/discriminator
stacked_gan = discriminator(gen_sample, reuse=True)

# Build Loss (Labels for real images: 1, for fake images: 0)
# Discriminator Loss for real and fake samples
disc_loss_real = tf.reduce_mean(tf.nn.sparse_softmax_cross_entro
py_with_logits(
    logits=disc_real, labels=tf.ones([batch_size], dtype=tf.int3
2)))
disc_loss_fake = tf.reduce_mean(tf.nn.sparse_softmax_cross_entro
py_with_logits(
    logits=disc_fake, labels=tf.zeros([batch_size], dtype=tf.int
32)))
# Sum both loss
disc_loss = disc_loss_real + disc_loss_fake
# Generator Loss (The generator tries to fool the discriminator,
 thus labels are 1)
gen_loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_wit
h_logits(
    logits=stacked_gan, labels=tf.ones([batch_size], dtype=tf.in
t32)))

# Build Optimizers
optimizer_gen = tf.train.AdamOptimizer(learning_rate=lr_generato
r, beta1=0.5, beta2=0.999)
optimizer_disc = tf.train.AdamOptimizer(learning_rate=lr_discrim
inator, beta1=0.5, beta2=0.999)

# Training Variables for each optimizer
# By default in TensorFlow, all variables are updated by each op
timizer, so we
```

```
# need to precise for each one of them the specific variables to
 update.
# Generator Network Variables
gen_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, s
cope='Generator')
# Discriminator Network Variables
disc_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
scope='Discriminator')

# Create training operations
# TensorFlow UPDATE_OPS collection holds all batch norm operatio
n to update the moving mean/stddev
gen_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS, scop
e='Generator')
# `control_dependencies` ensure that the `gen_update_ops` will b
e run before the `minimize` op (backprop)
with tf.control_dependencies(gen_update_ops):
    train_gen = optimizer_gen.minimize(gen_loss, var_list=gen_va
rs)
disc_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS, sco
pe='Discriminator')
with tf.control_dependencies(disc_update_ops):
    train_disc = optimizer_disc.minimize(disc_loss, var_list=dis
c_vars)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

```
# Start Training
# Start a new TF session
sess = tf.Session()

# Run the initializer
sess.run(init)

# Training
for i in range(1, num_steps+1):

    # Prepare Input Data
    # Get the next batch of MNIST data (only images are needed,
not labels)
    batch_x, _ = mnist.train.next_batch(batch_size)
    batch_x = np.reshape(batch_x, newshape=[-1, 28, 28, 1])
    # Rescale to [-1, 1], the input range of the discriminator
    batch_x = batch_x * 2. - 1.

    # Discriminator Training
    # Generate noise to feed to the generator
    z = np.random.uniform(-1., 1., size=[batch_size, noise_dim])
    _, dl = sess.run([train_disc, disc_loss], feed_dict={real_im
age_input: batch_x, noise_input: z, is_training:True})

    # Generator Training
    # Generate noise to feed to the generator
    z = np.random.uniform(-1., 1., size=[batch_size, noise_dim])
    _, gl = sess.run([train_gen, gen_loss], feed_dict={noise_inp
ut: z, is_training:True})

    if i % 500 == 0 or i == 1:
        print('Step %i: Generator Loss: %f, Discriminator Loss:
%f' % (i, gl, dl))
```

```
Step 1: Generator Loss: 3.590350, Discriminator Loss: 1.907586
Step 500: Generator Loss: 1.254698, Discriminator Loss: 1.005236
Step 1000: Generator Loss: 1.730409, Discriminator Loss: 0.83768
4
Step 1500: Generator Loss: 1.962198, Discriminator Loss: 0.61882
7
Step 2000: Generator Loss: 2.767945, Discriminator Loss: 0.37807
1
Step 2500: Generator Loss: 2.370605, Discriminator Loss: 0.56124
7
Step 3000: Generator Loss: 3.427798, Discriminator Loss: 0.40295
1
Step 3500: Generator Loss: 4.904454, Discriminator Loss: 0.55485
6
Step 4000: Generator Loss: 4.045284, Discriminator Loss: 0.45497
0
Step 4500: Generator Loss: 4.577699, Discriminator Loss: 0.68719
5
Step 5000: Generator Loss: 3.476081, Discriminator Loss: 0.21049
2
Step 5500: Generator Loss: 3.898139, Discriminator Loss: 0.14335
2
Step 6000: Generator Loss: 4.089877, Discriminator Loss: 1.08256
1
Step 6500: Generator Loss: 5.911457, Discriminator Loss: 0.15405
9
Step 7000: Generator Loss: 3.594872, Discriminator Loss: 0.15297
0
Step 7500: Generator Loss: 6.067883, Discriminator Loss: 0.08486
4
Step 8000: Generator Loss: 6.737456, Discriminator Loss: 0.40256
6
Step 8500: Generator Loss: 6.630128, Discriminator Loss: 0.03483
8
Step 9000: Generator Loss: 6.480587, Discriminator Loss: 0.42741
9
Step 9500: Generator Loss: 7.200409, Discriminator Loss: 0.12426
8
Step 10000: Generator Loss: 5.479313, Discriminator Loss: 0.1913
89
```

```
# Testing
# Generate images from noise, using the generator network.
n = 6
canvas = np.empty((28 * n, 28 * n))
for i in range(n):
    # Noise input.
    z = np.random.uniform(-1., 1., size=[n, noise_dim])
    # Generate image from noise.
    g = sess.run(gen_sample, feed_dict={noise_input: z, is_train
ing:False})
    # Rescale values to the original [0, 1] (from tanh -> [-1, 1
])
    g = (g + 1.) / 2.
    # Reverse colours for better display
    g = -1 * (g - 1)
    for j in range(n):
        # Draw the generated digits
        canvas[i * 28:(i + 1) * 28, j * 28:(j + 1) * 28] = g[j].
reshape([28, 28])

plt.figure(figsize=(n, n))
plt.imshow(canvas, origin="upper", cmap="gray")
plt.show()
```



Utilities

Save & Restore a Model

Save and Restore a model using TensorFlow. This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/).

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

import tensorflow as tf
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.001
batch_size = 100
display_step = 1
model_path = "/tmp/model.ckpt"

# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])


# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Hidden layer with RELU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['
b2'])
```

```
    layer_2 = tf.nn.relu(layer_2)
    # Output layer with linear activation
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out'
]
    return out_layer

# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])
),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes])
)
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Construct model
pred = multilayer_perceptron(x, weights, biases)

# Define loss and optimizer
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(lo
gits=pred, labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).
minimize(cost)

# Initializing the variables
init = tf.global_variables_initializer()
```

```
# 'Saver' op to save and restore all the variables
saver = tf.train.Saver()
```

```
# Running first session
print("Starting 1st session...")
with tf.Session() as sess:
    # Initialize variables
    sess.run(init)

    # Training cycle
    for epoch in range(3):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size
)
            # Run optimization op (backprop) and cost op (to get
 loss value)
            _, c = sess.run([optimizer, cost], feed_dict={x: bat
ch_x,
                                                          y: bat
ch_y})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(avg_cost)
    print("First Optimization Finished!")

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(
y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"
))
    print("Accuracy:", accuracy.eval({x: mnist.test.images, y: m
nist.test.labels}))

    # Save model weights to disk
    save_path = saver.save(sess, model_path)
    print("Model saved in file: %s" % save_path)
```

```
Starting 1st session...
Epoch: 0001 cost= 187.778896380
Epoch: 0002 cost= 42.367902536
Epoch: 0003 cost= 26.488964058
First Optimization Finished!
Accuracy: 0.9075
Model saved in file: /tmp/model.ckpt
```

```
# Running a new session
print("Starting 2nd session...")
with tf.Session() as sess:
    # Initialize variables
    sess.run(init)

    # Restore model weights from previously saved model
    load_path = saver.restore(sess, model_path)
    print("Model restored from file: %s" % save_path)

    # Resume training
    for epoch in range(7):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples / batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_x, batch_y = mnist.train.next_batch(batch_size
)
            # Run optimization op (backprop) and cost op (to get
 loss value)
            _, c = sess.run([optimizer, cost], feed_dict={x: bat
ch_x,
                                                          y: bat
ch_y})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print("Epoch:", '%04d' % (epoch + 1), "cost=", \
                "{:.9f}".format(avg_cost))
    print("Second Optimization Finished!")

    # Test model
    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(
y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"
))
    print("Accuracy:", accuracy.eval(
        {x: mnist.test.images, y: mnist.test.labels}))
```

```
Starting 2nd session...
Model restored from file: /tmp/model.ckpt
Epoch: 0001 cost= 18.292712951
Epoch: 0002 cost= 13.404136196
Epoch: 0003 cost= 9.855191723
Epoch: 0004 cost= 7.276933088
Epoch: 0005 cost= 5.564581285
Epoch: 0006 cost= 4.165259939
Epoch: 0007 cost= 3.139393926
Second Optimization Finished!
Accuracy: 0.9385
```

Tensorboard Basics

Graph and Loss visualization using Tensorboard. This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/).

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

import tensorflow as tf

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

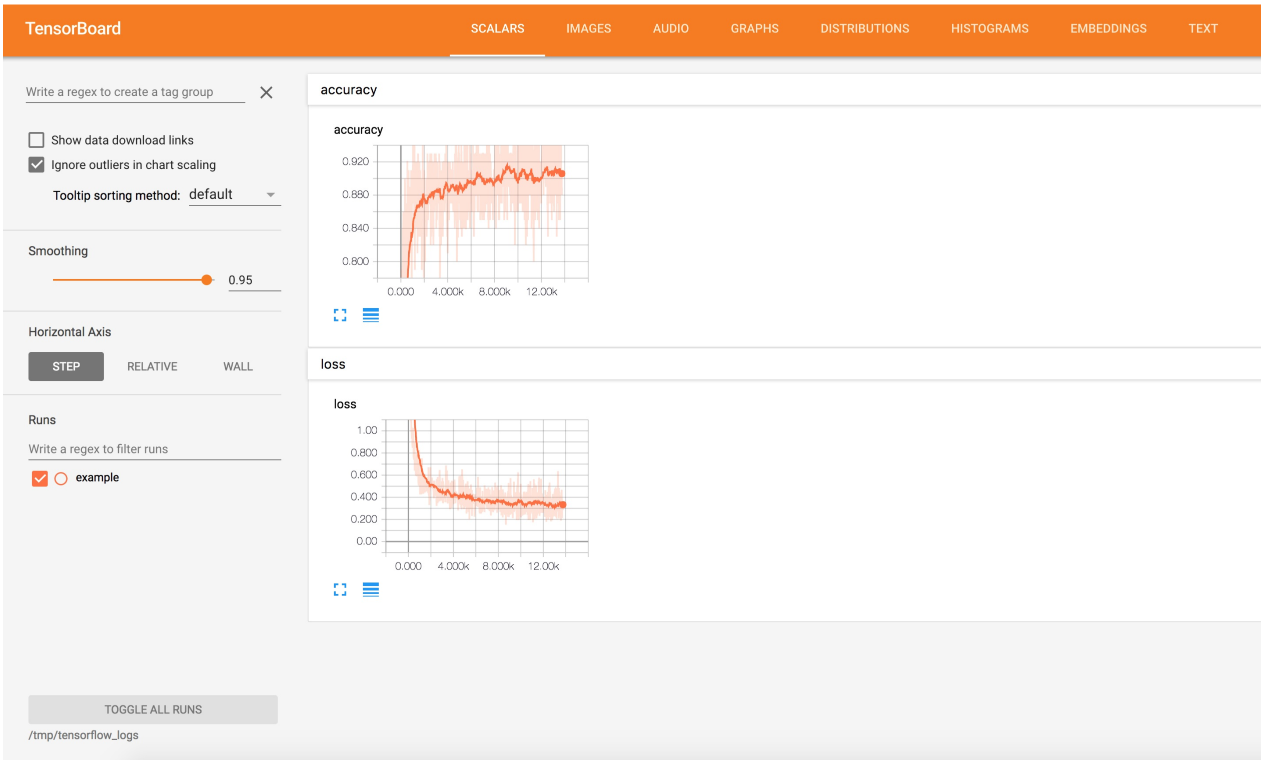
```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_epoch = 1
logs_path = '/tmp/tensorflow_logs/example/'

# tf Graph Input
# mnist data image of shape 28*28=784
x = tf.placeholder(tf.float32, [None, 784], name='InputData')
# 0-9 digits recognition => 10 classes
y = tf.placeholder(tf.float32, [None, 10], name='LabelData')

# Set model weights
W = tf.Variable(tf.zeros([784, 10]), name='Weights')
b = tf.Variable(tf.zeros([10]), name='Bias')
```

```
# Construct model and encapsulating all ops into scopes, making
# Tensorboard's Graph visualization more convenient
with tf.name_scope('Model'):
    # Model
    pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax
with tf.name_scope('Loss'):
    # Minimize error using cross entropy
    cost = tf.reduce_mean(-tf.reduce_sum(y * tf.log(pred), reduc
tion_indices=1))
with tf.name_scope('SGD'):
    # Gradient Descent
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
.minimize(cost)
with tf.name_scope('Accuracy'):
    # Accuracy
    acc = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    acc = tf.reduce_mean(tf.cast(acc, tf.float32))

# Initializing the variables
init = tf.global_variables_initializer()

# Create a summary to monitor cost tensor
tf.summary.scalar("loss", cost)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("accuracy", acc)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
```

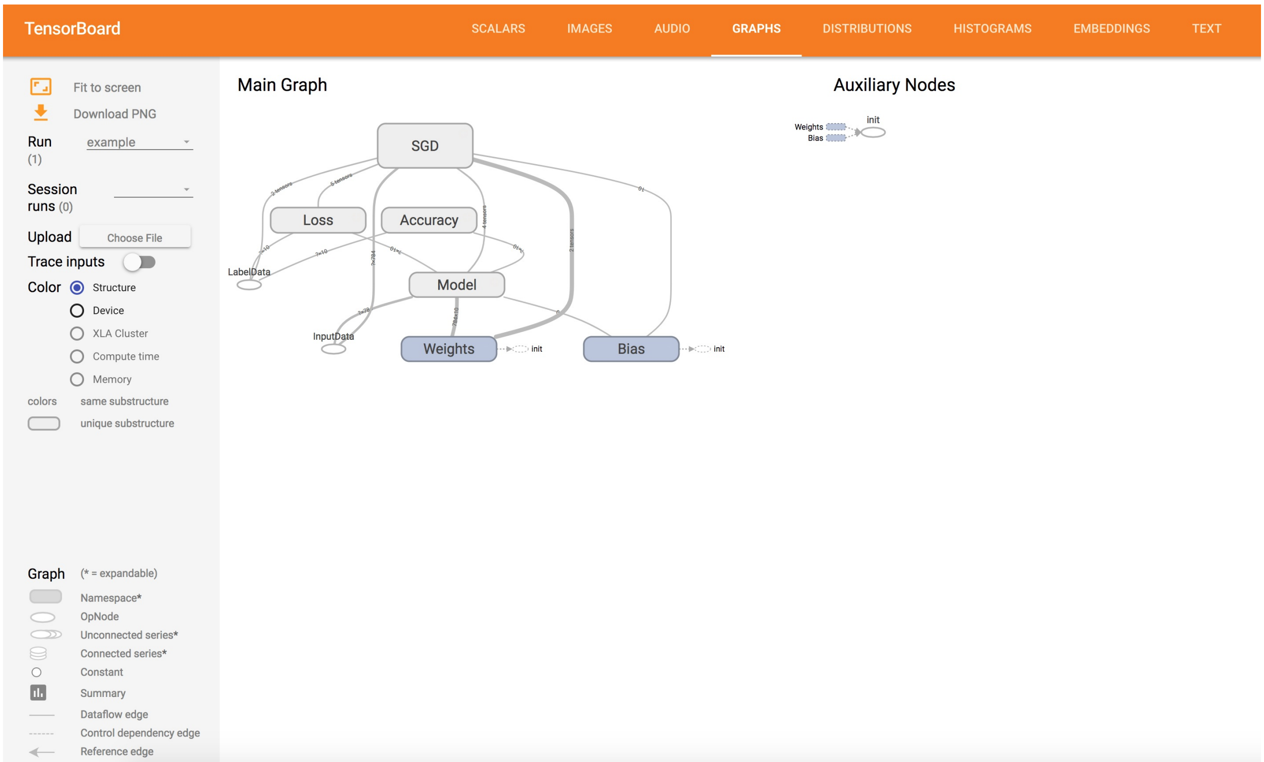
```
# Start Training
with tf.Session() as sess:
    sess.run(init)

    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logs_path, graph=tf.g
et_default_graph())

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples / batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_si
ze)
            # Run optimization op (backprop), cost op (to get lo
ss value)
            # and summary nodes
            _, c, summary = sess.run([optimizer, cost, merged_su
mmary_op],
                                     feed_dict={x: batch_xs, y:
batch_ys})
            # Write logs at every iteration
            summary_writer.add_summary(summary, epoch * total_ba
tch + i)
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if (epoch+1) % display_epoch == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}"
.format(avg_cost))

    print("Optimization Finished!")

    # Test model
    # Calculate accuracy
    print("Accuracy:", acc.eval({x: mnist.test.images, y: mnist.
test.labels}))

    print("Run the command line:\n" \
          "--> tensorboard --logdir=/tmp/tensorflow_logs " \
          "\nThen open http://0.0.0.0:6006/ into your web browse
r")
```

```
Epoch: 0001 cost= 1.182138961
Epoch: 0002 cost= 0.664609327
Epoch: 0003 cost= 0.552565036
Epoch: 0004 cost= 0.498541865
Epoch: 0005 cost= 0.465393374
Epoch: 0006 cost= 0.442491178
Epoch: 0007 cost= 0.425474149
Epoch: 0008 cost= 0.412152022
Epoch: 0009 cost= 0.401320939
Epoch: 0010 cost= 0.392305281
Epoch: 0011 cost= 0.384732356
Epoch: 0012 cost= 0.378109478
Epoch: 0013 cost= 0.372409370
Epoch: 0014 cost= 0.367236996
Epoch: 0015 cost= 0.362727492
Epoch: 0016 cost= 0.358627345
Epoch: 0017 cost= 0.354815522
Epoch: 0018 cost= 0.351413656
Epoch: 0019 cost= 0.348314827
Epoch: 0020 cost= 0.345429416
Epoch: 0021 cost= 0.342749324
Epoch: 0022 cost= 0.340224642
Epoch: 0023 cost= 0.337897302
Epoch: 0024 cost= 0.335720168
Epoch: 0025 cost= 0.333691911
Optimization Finished!
Accuracy: 0.9143
Run the command line:
--> tensorboard --logdir=/tmp/tensorflow_logs
Then open http://0.0.0.0:6006/ into your web browser
```

Loss and Accuracy Visualization



Graph Visualization



Tensorboard Advanced

Advanced visualization using Tensorboard (weights, gradient, ...). This example is using the MNIST database of handwritten digits (http://yann.lecun.com/exdb/mnist/).

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

import tensorflow as tf

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1
logs_path = '/tmp/tensorflow_logs/example/'

# Network Parameters
n_hidden_1 = 256 # 1st layer number of features
n_hidden_2 = 256 # 2nd layer number of features
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph Input
# mnist data image of shape 28*28=784
x = tf.placeholder(tf.float32, [None, 784], name='InputData')
# 0-9 digits recognition => 10 classes
y = tf.placeholder(tf.float32, [None, 10], name='LabelData')
```

```
# Create model
def multilayer_perceptron(x, weights, biases):
    # Hidden layer with RELU activation
    layer_1 = tf.add(tf.matmul(x, weights['w1']), biases['b1'])
    layer_1 = tf.nn.relu(layer_1)
    # Create a summary to visualize the first layer ReLU activat
ion
    tf.summary.histogram("relu1", layer_1)
    # Hidden layer with RELU activation
    layer_2 = tf.add(tf.matmul(layer_1, weights['w2']), biases['
b2'])
    layer_2 = tf.nn.relu(layer_2)
    # Create another summary to visualize the second layer ReLU
activation
    tf.summary.histogram("relu2", layer_2)
    # Output layer
    out_layer = tf.add(tf.matmul(layer_2, weights['w3']), biases[
'b3'])
    return out_layer

# Store layers weight & bias
weights = {
    'w1': tf.Variable(tf.random_normal([n_input, n_hidden_1]), n
ame='W1'),
    'w2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])
, name='W2'),
    'w3': tf.Variable(tf.random_normal([n_hidden_2, n_classes]),
 name='W3')
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1]), name='b1')
,
    'b2': tf.Variable(tf.random_normal([n_hidden_2]), name='b2')
,
    'b3': tf.Variable(tf.random_normal([n_classes]), name='b3')
}
```

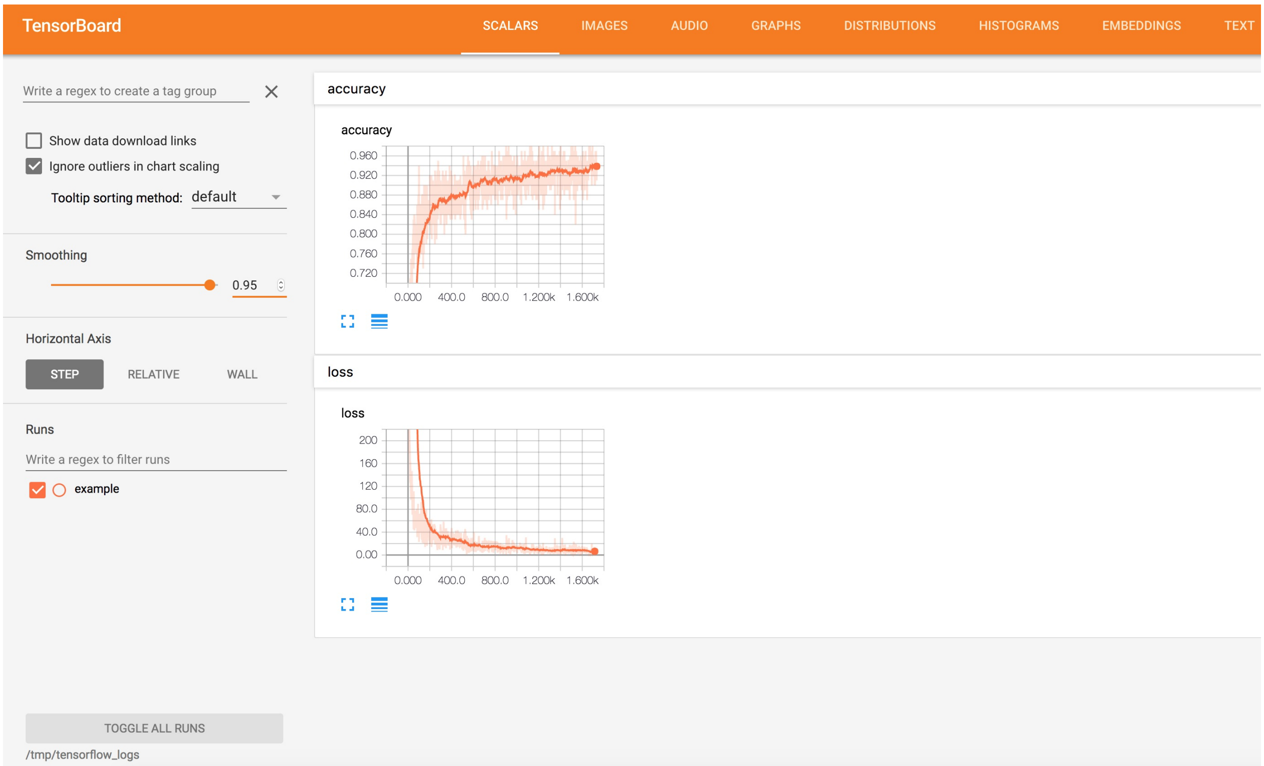
```
# Encapsulating all ops into scopes, making Tensorboard's Graph
# Visualization more convenient
with tf.name_scope('Model'):
    # Build model
    pred = multilayer_perceptron(x, weights, biases)

with tf.name_scope('Loss'):
    # Softmax Cross entropy (cost function)
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logit
s(logits=pred, labels=y))

with tf.name_scope('SGD'):
    # Gradient Descent
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    # Op to calculate every variable gradient
    grads = tf.gradients(loss, tf.trainable_variables())
    grads = list(zip(grads, tf.trainable_variables()))
    # Op to update all variables according to their gradient
    apply_grads = optimizer.apply_gradients(grads_and_vars=grads
)

with tf.name_scope('Accuracy'):
    # Accuracy
    acc = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
    acc = tf.reduce_mean(tf.cast(acc, tf.float32))
```

```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Create a summary to monitor cost tensor
tf.summary.scalar("loss", loss)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("accuracy", acc)
# Create summaries to visualize weights
for var in tf.trainable_variables():
    tf.summary.histogram(var.name, var)
# Summarize all gradients
for grad, var in grads:
    tf.summary.histogram(var.name + '/gradient', grad)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()
```

```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logs_path,
                                           graph=tf.get_default
_graph())

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_si
ze)
            # Run optimization op (backprop), cost op (to get lo
ss value)
            # and summary nodes
            _, c, summary = sess.run([apply_grads, loss, merged_
summary_op],
                                     feed_dict={x: batch_xs, y:
batch_ys})
            # Write logs at every iteration
            summary_writer.add_summary(summary, epoch * total_ba
tch + i)
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}"
.format(avg_cost))

    print("Optimization Finished!")

    # Test model
    # Calculate accuracy
    print("Accuracy:", acc.eval({x: mnist.test.images, y: mnist.
test.labels}))

    print("Run the command line:\n" \
          "--> tensorboard --logdir=/tmp/tensorflow_logs " \
          "\nThen open http://0.0.0.0:6006/ into your web browse
r")
```

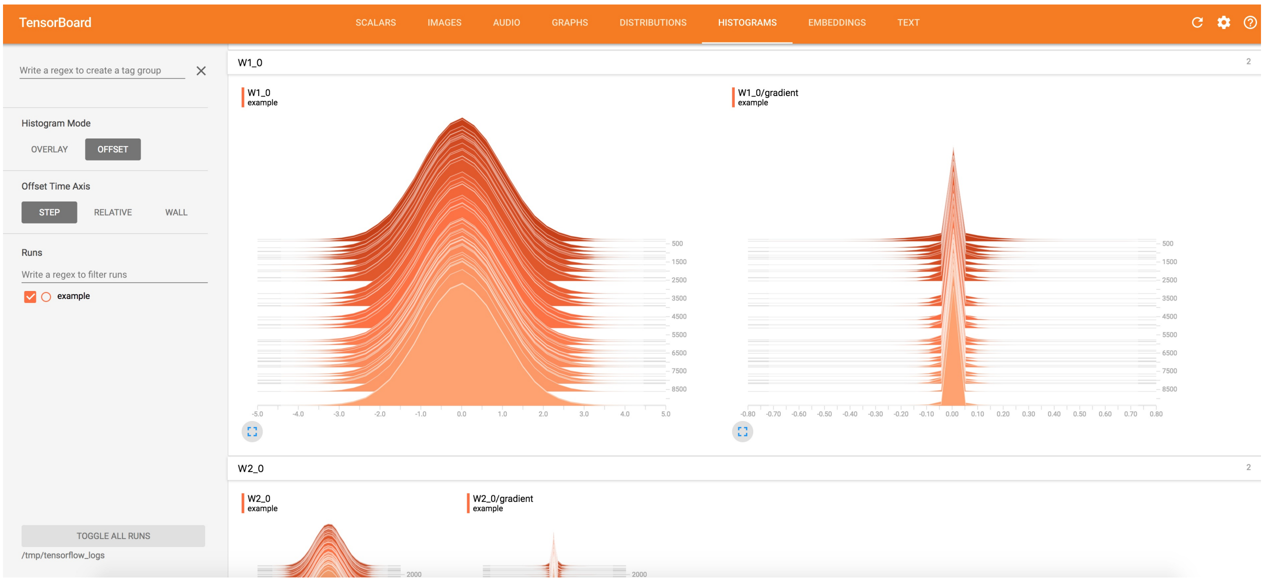
```
Epoch: 0001 cost= 59.570364205
Epoch: 0002 cost= 13.585465186
Epoch: 0003 cost= 8.379069252
Epoch: 0004 cost= 6.005265894
Epoch: 0005 cost= 4.498054792
Epoch: 0006 cost= 3.503682522
Epoch: 0007 cost= 2.822272765
Epoch: 0008 cost= 2.306899852
Epoch: 0009 cost= 1.912765543
Epoch: 0010 cost= 1.597006118
Epoch: 0011 cost= 1.330172869
Epoch: 0012 cost= 1.142490618
Epoch: 0013 cost= 0.939443911
Epoch: 0014 cost= 0.820920588
Epoch: 0015 cost= 0.702543302
Epoch: 0016 cost= 0.604815631
Epoch: 0017 cost= 0.505682561
Epoch: 0018 cost= 0.439700446
Epoch: 0019 cost= 0.378268929
Epoch: 0020 cost= 0.299557848
Epoch: 0021 cost= 0.269859066
Epoch: 0022 cost= 0.230899029
Epoch: 0023 cost= 0.183722090
Epoch: 0024 cost= 0.164173368
Epoch: 0025 cost= 0.142141250
Optimization Finished!
Accuracy: 0.9336
Run the command line:
--> tensorboard --logdir=/tmp/tensorflow_logs
Then open http://0.0.0.0:6006/ into your web browser
```

Loss and Accuracy Visualization

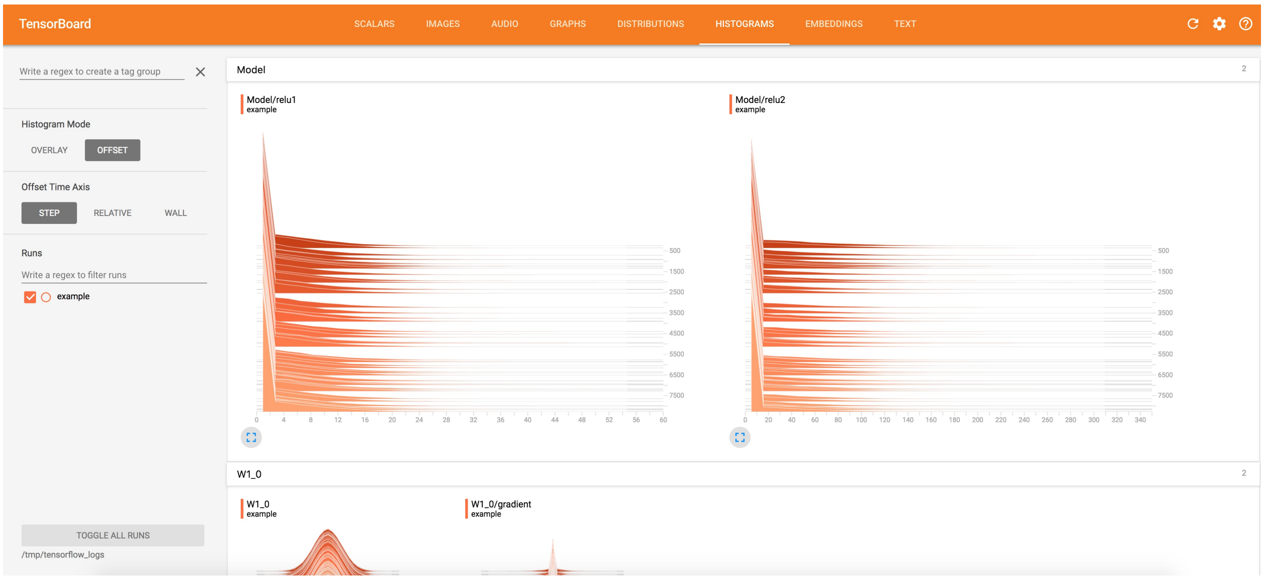


Computation Graph Visualization

Weights and Gradients Visualization



Activations Visualization



Data Management

Build an Image Dataset in TensorFlow.

For this example, you need to make your own set of images (JPEG). We will show 2 different ways to build that dataset:

- From a root folder, that will have a sub-folder containing images for each class

```
ROOT_FOLDER
   |-------- SUBFOLDER (CLASS 0)
   |              |
   |              | ----- image1.jpg
   |              | ----- image2.jpg
   |              | ----- etc...
   |
   |-------- SUBFOLDER (CLASS 1)
   |              |
   |              | ----- image1.jpg
   |              | ----- image2.jpg
   |              | ----- etc...
```

- From a plain text file, that will list all images with their class ID:

```
/path/to/image/1.jpg CLASS_ID
/path/to/image/2.jpg CLASS_ID
/path/to/image/3.jpg CLASS_ID
/path/to/image/4.jpg CLASS_ID
etc...
```

Below, there are some parameters that you need to change (Marked 'CHANGE HERE'), such as the dataset path.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
from __future__ import print_function

import tensorflow as tf
import os

# Dataset Parameters - CHANGE HERE
MODE = 'folder' # or 'file', if you choose a plain text file (se
e above).
DATASET_PATH = '/path/to/dataset/' # the dataset file or root fo
lder path.

# Image Parameters
N_CLASSES = 2 # CHANGE HERE, total number of classes
IMG_HEIGHT = 64 # CHANGE HERE, the image height to be resized to
IMG_WIDTH = 64 # CHANGE HERE, the image width to be resized to
CHANNELS = 3 # The 3 color channels, change to 1 if grayscale
```

```
# Reading the dataset
# 2 modes: 'file' or 'folder'
def read_images(dataset_path, mode, batch_size):
    imagepaths, labels = list(), list()
    if mode == 'file':
        # Read dataset file
        data = open(dataset_path, 'r').read().splitlines()
        for d in data:
            imagepaths.append(d.split(' ')[0])
            labels.append(int(d.split(' ')[1]))
    elif mode == 'folder':
        # An ID will be affected to each sub-folders by alphabet
ical order
        label = 0
        # List the directory
        try:  # Python 2
            classes = sorted(os.walk(dataset_path).next()[1])
        except Exception:  # Python 3
            classes = sorted(os.walk(dataset_path).__next__()[1]
)
        # List each sub-directory (the classes)
        for c in classes:
            c_dir = os.path.join(dataset_path, c)
            try:  # Python 2
                walk = os.walk(c_dir).next()
            except Exception:  # Python 3
                walk = os.walk(c_dir).__next__()
            # Add each image to the training set
            for sample in walk[2]:
                # Only keeps jpeg images
                if sample.endswith('.jpg') or sample.endswith('.
jpeg'):
                    imagepaths.append(os.path.join(c_dir, sample
```

```
))
                    labels.append(label)
            label += 1
    else:
        raise Exception("Unknown mode.")

    # Convert to Tensor
    imagepaths = tf.convert_to_tensor(imagepaths, dtype=tf.strin
g)
    labels = tf.convert_to_tensor(labels, dtype=tf.int32)
    # Build a TF Queue, shuffle data
    image, label = tf.train.slice_input_producer([imagepaths, la
bels],
                                                 shuffle=True)

    # Read images from disk
    image = tf.read_file(image)
    image = tf.image.decode_jpeg(image, channels=CHANNELS)

    # Resize images to a common size
    image = tf.image.resize_images(image, [IMG_HEIGHT, IMG_WIDTH
])

    # Normalize
    image = image * 1.0/127.5 - 1.0

    # Create batches
    X, Y = tf.train.batch([image, label], batch_size=batch_size,
                          capacity=batch_size * 8,
                          num_threads=4)

    return X, Y
```

```
# -----------------------------------------------
# THIS IS A CLASSIC CNN (see examples, section 3)
# -----------------------------------------------
# Note that a few elements have changed (usage of queues).

# Parameters
learning_rate = 0.001
num_steps = 10000
batch_size = 128
display_step = 100

# Network Parameters
dropout = 0.75 # Dropout, probability to keep units

# Build the data input
X, Y = read_images(DATASET_PATH, MODE, batch_size)
```

```
# Create model
def conv_net(x, n_classes, dropout, reuse, is_training):
    # Define a scope for reusing the variables
    with tf.variable_scope('ConvNet', reuse=reuse):

        # Convolution Layer with 32 filters and a kernel size of
 5
        conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu
)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        conv1 = tf.layers.max_pooling2d(conv1, 2, 2)

        # Convolution Layer with 32 filters and a kernel size of
 5
        conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.
relu)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        conv2 = tf.layers.max_pooling2d(conv2, 2, 2)

        # Flatten the data to a 1-D vector for the fully connect
ed layer
        fc1 = tf.contrib.layers.flatten(conv2)

        # Fully connected layer (in contrib folder for now)
        fc1 = tf.layers.dense(fc1, 1024)
        # Apply Dropout (if is_training is False, dropout is not
 applied)
        fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_t
raining)

        # Output layer, class prediction
        out = tf.layers.dense(fc1, n_classes)
        # Because 'softmax_cross_entropy_with_logits' already ap
ply softmax,
        # we only apply softmax to testing network
        out = tf.nn.softmax(out) if not is_training else out

    return out
```

```
# Because Dropout have different behavior at training and predic
tion time, we
# need to create 2 distinct computation graphs that share the sa
me weights.

# Create a graph for training
logits_train = conv_net(X, N_CLASSES, dropout, reuse=False, is_t
raining=True)
# Create another graph for testing that reuse the same weights
```

```
logits_test = conv_net(X, N_CLASSES, dropout, reuse=True, is_tra
ining=False)

# Define loss and optimizer (with train logits, for dropout to t
ake effect)
loss_op = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with
_logits(
    logits=logits_train, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits_test, 1), tf.cast(Y, tf
.int64))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Saver object
saver = tf.train.Saver()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    # Start the data queue
    tf.train.start_queue_runners()

    # Training cycle
    for step in range(1, num_steps+1):

        if step % display_step == 0:
            # Run optimization and calculate batch loss and accu
racy
            _, loss, acc = sess.run([train_op, loss_op, accuracy
])
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= "
 + \
                  "{:.3f}".format(acc))
        else:
            # Only run the optimization op (backprop)
            sess.run(train_op)

    print("Optimization Finished!")

    # Save your model
    saver.save(sess, 'my_tf_model')
```

TensorFlow Dataset API

In this example, we will show how to load numpy array data into the new TensorFlow 'Dataset' API. The Dataset API implements an optimized data pipeline with queues, that make data processing and training faster (especially on GPU).

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
import tensorflow as tf

# Import MNIST data (Numpy format)
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Parameters
learning_rate = 0.01
num_steps = 1000
batch_size = 128
display_step = 100

# Network Parameters
n_input = 784 # MNIST data input (img shape: 28*28)
n_classes = 10 # MNIST total classes (0-9 digits)
dropout = 0.75 # Dropout, probability to keep units

sess = tf.Session()

# Create a dataset tensor from the images and the labels
dataset = tf.contrib.data.Dataset.from_tensor_slices(
    (mnist.train.images, mnist.train.labels))
# Create batches of data
dataset = dataset.batch(batch_size)
# Create an iterator, to go over the dataset
iterator = dataset.make_initializable_iterator()
# It is better to use 2 placeholders, to avoid to load all data
into memory,
# and avoid the 2Gb restriction length of a tensor.
_data = tf.placeholder(tf.float32, [None, n_input])
_labels = tf.placeholder(tf.float32, [None, n_classes])
# Initialize the iterator
sess.run(iterator.initializer, feed_dict={_data: mnist.train.ima
ges,
                                          _labels: mnist.train.l
abels})

# Neural Net Input
X, Y = iterator.get_next()
```

```
# -----------------------------------------------
# THIS IS A CLASSIC CNN (see examples, section 3)
# -----------------------------------------------
# Note that a few elements have changed (usage of sess run).

# Create model
def conv_net(x, n_classes, dropout, reuse, is_training):
    # Define a scope for reusing the variables
    with tf.variable_scope('ConvNet', reuse=reuse):
        # MNIST data input is a 1-D vector of 784 features (28*2
8 pixels)
        # Reshape to match picture format [Height x Width x Chan
nel]
        # Tensor input become 4-D: [Batch Size, Height, Width, C
hannel]
        x = tf.reshape(x, shape=[-1, 28, 28, 1])
```

```
        # Convolution Layer with 32 filters and a kernel size of
 5
        conv1 = tf.layers.conv2d(x, 32, 5, activation=tf.nn.relu
)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        conv1 = tf.layers.max_pooling2d(conv1, 2, 2)

        # Convolution Layer with 32 filters and a kernel size of
 5
        conv2 = tf.layers.conv2d(conv1, 64, 3, activation=tf.nn.
relu)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        conv2 = tf.layers.max_pooling2d(conv2, 2, 2)

        # Flatten the data to a 1-D vector for the fully connect
ed layer
        fc1 = tf.contrib.layers.flatten(conv2)

        # Fully connected layer (in contrib folder for now)
        fc1 = tf.layers.dense(fc1, 1024)
        # Apply Dropout (if is_training is False, dropout is not
 applied)
        fc1 = tf.layers.dropout(fc1, rate=dropout, training=is_t
raining)

        # Output layer, class prediction
        out = tf.layers.dense(fc1, n_classes)
        # Because 'softmax_cross_entropy_with_logits' already ap
ply softmax,
        # we only apply softmax to testing network
        out = tf.nn.softmax(out) if not is_training else out

    return out


# Because Dropout have different behavior at training and predic
tion time, we
# need to create 2 distinct computation graphs that share the sa
me weights.

# Create a graph for training
logits_train = conv_net(X, n_classes, dropout, reuse=False, is_t
raining=True)
# Create another graph for testing that reuse the same weights,
but has
# different behavior for 'dropout' (not applied).
logits_test = conv_net(X, n_classes, dropout, reuse=True, is_tra
ining=False)

# Define loss and optimizer (with train logits, for dropout to t
```

```
ake effect)
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits
(
    logits=logits_train, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits_test, 1), tf.argmax(Y, 1
))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

```
# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Run the initializer
sess.run(init)

# Training cycle
for step in range(1, num_steps + 1):

    try:
        # Run optimization
        sess.run(train_op)
    except tf.errors.OutOfRangeError:
        # Reload the iterator when it reaches the end of the dat
aset
        sess.run(iterator.initializer,
                 feed_dict={_data: mnist.train.images,
                            _labels: mnist.train.labels})
        sess.run(train_op)

    if step % display_step == 0 or step == 1:
        # Calculate batch loss and accuracy
        # (note that this consume a new batch of data)
        loss, acc = sess.run([loss_op, accuracy])
        print("Step " + str(step) + ", Minibatch Loss= " + \
              "{:.4f}".format(loss) + ", Training Accuracy= " +
\
              "{:.3f}".format(acc))

print("Optimization Finished!")
```

```
Step 1, Minibatch Loss= 7.9429, Training Accuracy= 0.070
Step 100, Minibatch Loss= 0.3491, Training Accuracy= 0.922
Step 200, Minibatch Loss= 0.2343, Training Accuracy= 0.922
Step 300, Minibatch Loss= 0.1838, Training Accuracy= 0.969
Step 400, Minibatch Loss= 0.1715, Training Accuracy= 0.953
Step 500, Minibatch Loss= 0.2730, Training Accuracy= 0.938
Step 600, Minibatch Loss= 0.3427, Training Accuracy= 0.953
Step 700, Minibatch Loss= 0.2261, Training Accuracy= 0.961
Step 800, Minibatch Loss= 0.1487, Training Accuracy= 0.953
Step 900, Minibatch Loss= 0.1438, Training Accuracy= 0.945
Step 1000, Minibatch Loss= 0.1786, Training Accuracy= 0.961
Optimization Finished!
```

Multi GPU

Multi-GPU Basics

Basic Multi-GPU computation example using TensorFlow library.

This tutorial requires your machine to have 2 GPUs "/cpu:0": The CPU of your machine. "/gpu:0": The first GPU of your machine "/gpu:1": The second GPU of your machine For this example, we are using 2 GTX-980

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

```
import numpy as np
import tensorflow as tf
import datetime
```

```
#Processing Units logs
log_device_placement = True

#num of multiplications to perform
n = 10
```

```
# Example: compute A^n + B^n on 2 GPUs

# Create random large matrix
A = np.random.rand(1e4, 1e4).astype('float32')
B = np.random.rand(1e4, 1e4).astype('float32')

# Creates a graph to store results
c1 = []
c2 = []

# Define matrix power
def matpow(M, n):
    if n < 1: #Abstract cases where n < 1
        return M
    else:
        return tf.matmul(M, matpow(M, n-1))
```

```
# Single GPU computing

with tf.device('/gpu:0'):
    a = tf.constant(A)
    b = tf.constant(B)
    #compute A^n and B^n and store results in c1
    c1.append(matpow(a, n))
    c1.append(matpow(b, n))

with tf.device('/cpu:0'):
  sum = tf.add_n(c1) #Addition of all elements in c1, i.e. A^n +
 B^n

t1_1 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(log_device_placement=log_d
evice_placement)) as sess:
    # Runs the op.
    sess.run(sum)
t2_1 = datetime.datetime.now()
```

```
# Multi GPU computing
# GPU:0 computes A^n
with tf.device('/gpu:0'):
    #compute A^n and store result in c2
    a = tf.constant(A)
    c2.append(matpow(a, n))

#GPU:1 computes B^n
with tf.device('/gpu:1'):
    #compute B^n and store result in c2
    b = tf.constant(B)
    c2.append(matpow(b, n))

with tf.device('/cpu:0'):
  sum = tf.add_n(c2) #Addition of all elements in c2, i.e. A^n +
 B^n

t1_2 = datetime.datetime.now()
with tf.Session(config=tf.ConfigProto(log_device_placement=log_d
evice_placement)) as sess:
    # Runs the op.
    sess.run(sum)
t2_2 = datetime.datetime.now()
```

```
print "Single GPU computation time: " + str(t2_1-t1_1)
print "Multi GPU computation time: " + str(t2_2-t1_2)
```

```
Single GPU computation time: 0:00:11.833497
Multi GPU computation time: 0:00:07.085913
```

Multi-GPU Training Example

Train a convolutional neural network on multiple GPU with TensorFlow.

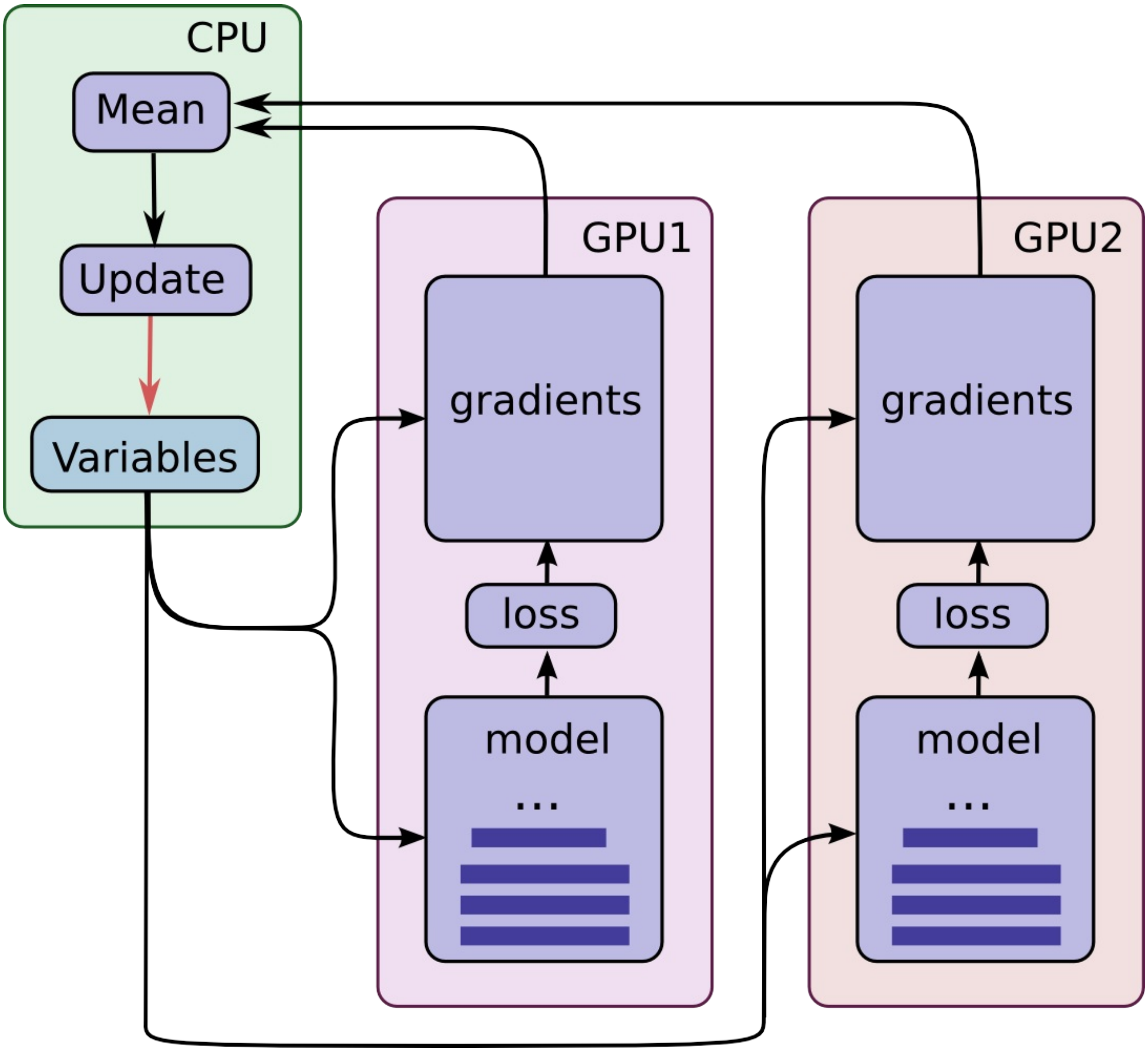
This example is using TensorFlow layers, see 'convolutional_network_raw' example for a raw TensorFlow implementation with variables.

- Author: Aymeric Damien
- Project: https://github.com/aymericdamien/TensorFlow-Examples/

Training with multiple GPU cards

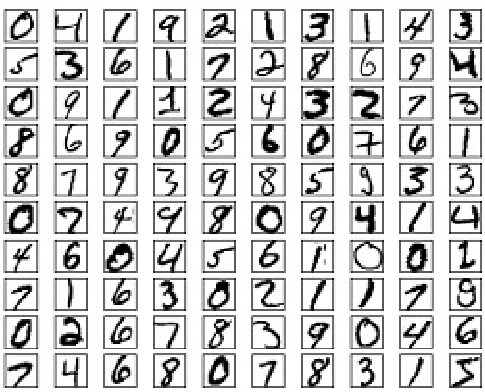
In this example, we are using data parallelism to split the training accross multiple GPUs. Each GPU has a full replica of the neural network model, and the weights (i.e. variables) are updated synchronously by waiting that each GPU process its batch of data.

First, each GPU process a distinct batch of data and compute the corresponding gradients, then, all gradients are accumulated in the CPU and averaged. The model weights are finally updated with the gradients averaged, and the new model weights are sent back to each GPU, to repeat the training process.



MNIST Dataset Overview

This example is using MNIST handwritten digits. The dataset contains 60,000 examples for training and 10,000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28x28 pixels) with values from 0 to 1. For simplicity, each image has been flatten and converted to a 1-D numpy array of 784 features (28*28).



More info: http://yann.lecun.com/exdb/mnist/

```
from __future__ import print_function

import numpy as np
import tensorflow as tf
import time

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Parameters
num_gpus = 2
num_steps = 200
learning_rate = 0.001
batch_size = 1024
display_step = 10

# Network Parameters
num_input = 784 # MNIST data input (img shape: 28*28)
num_classes = 10 # MNIST total classes (0-9 digits)
dropout = 0.75 # Dropout, probability to keep units
```

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

```
# Build a convolutional neural network
def conv_net(x, n_classes, dropout, reuse, is_training):
    # Define a scope for reusing the variables
    with tf.variable_scope('ConvNet', reuse=reuse):
        # MNIST data input is a 1-D vector of 784 features (28*2
8 pixels)
        # Reshape to match picture format [Height x Width x Chan
nel]
        # Tensor input become 4-D: [Batch Size, Height, Width, C
hannel]
        x = tf.reshape(x, shape=[-1, 28, 28, 1])

        # Convolution Layer with 64 filters and a kernel size of
 5
        x = tf.layers.conv2d(x, 64, 5, activation=tf.nn.relu)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        x = tf.layers.max_pooling2d(x, 2, 2)

        # Convolution Layer with 256 filters and a kernel size o
```

```
f 5
        x = tf.layers.conv2d(x, 256, 3, activation=tf.nn.relu)
        # Convolution Layer with 512 filters and a kernel size o
f 5
        x = tf.layers.conv2d(x, 512, 3, activation=tf.nn.relu)
        # Max Pooling (down-sampling) with strides of 2 and kern
el size of 2
        x = tf.layers.max_pooling2d(x, 2, 2)

        # Flatten the data to a 1-D vector for the fully connect
ed layer
        x = tf.contrib.layers.flatten(x)

        # Fully connected layer (in contrib folder for now)
        x = tf.layers.dense(x, 2048)
        # Apply Dropout (if is_training is False, dropout is not
 applied)
        x = tf.layers.dropout(x, rate=dropout, training=is_train
ing)

        # Fully connected layer (in contrib folder for now)
        x = tf.layers.dense(x, 1024)
        # Apply Dropout (if is_training is False, dropout is not
 applied)
        x = tf.layers.dropout(x, rate=dropout, training=is_train
ing)

        # Output layer, class prediction
        out = tf.layers.dense(x, n_classes)
        # Because 'softmax_cross_entropy_with_logits' loss alrea
dy apply
        # softmax, we only apply softmax to testing network
        out = tf.nn.softmax(out) if not is_training else out

    return out
```

```
# Build the function to average the gradients
def average_gradients(tower_grads):
    average_grads = []
    for grad_and_vars in zip(*tower_grads):
        # Note that each grad_and_vars looks like the following:
        #   ((grad0_gpu0, var0_gpu0), ... , (grad0_gpuN, var0_gp
uN))
        grads = []
        for g, _ in grad_and_vars:
            # Add 0 dimension to the gradients to represent the
tower.
            expanded_g = tf.expand_dims(g, 0)

            # Append on a 'tower' dimension which we will averag
e over below.
            grads.append(expanded_g)

        # Average over the 'tower' dimension.
        grad = tf.concat(grads, 0)
        grad = tf.reduce_mean(grad, 0)

        # Keep in mind that the Variables are redundant because
they are shared
        # across towers. So .. we will just return the first tow
er's pointer to
        # the Variable.
        v = grad_and_vars[0][1]
        grad_and_var = (grad, v)
        average_grads.append(grad_and_var)
    return average_grads
```

```
# By default, all variables will be placed on '/gpu:0'
# So we need a custom device function, to assign all variables t
o '/cpu:0'
# Note: If GPUs are peered, '/gpu:0' can be a faster option
PS_OPS = ['Variable', 'VariableV2', 'AutoReloadVariable']

def assign_to_device(device, ps_device='/cpu:0'):
    def _assign(op):
        node_def = op if isinstance(op, tf.NodeDef) else op.node
_def
        if node_def.op in PS_OPS:
            return "/" + ps_device
        else:
            return device

    return _assign
```

```
# Place all ops on CPU by default
```

```
with tf.device('/cpu:0'):
    tower_grads = []
    reuse_vars = False

    # tf Graph input
    X = tf.placeholder(tf.float32, [None, num_input])
    Y = tf.placeholder(tf.float32, [None, num_classes])

    # Loop over all GPUs and construct their own computation gra
ph
    for i in range(num_gpus):
        with tf.device(assign_to_device('/gpu:{}'.format(i), ps_
device='/cpu:0')):

            # Split data between GPUs
            _x = X[i * batch_size: (i+1) * batch_size]
            _y = Y[i * batch_size: (i+1) * batch_size]

            # Because Dropout have different behavior at trainin
g and prediction time, we
            # need to create 2 distinct computation graphs that
share the same weights.

            # Create a graph for training
            logits_train = conv_net(_x, num_classes, dropout,
                                    reuse=reuse_vars, is_trainin
g=True)
            # Create another graph for testing that reuse the sa
me weights
            logits_test = conv_net(_x, num_classes, dropout,
                                   reuse=True, is_training=False
)

            # Define loss and optimizer (with train logits, for
dropout to take effect)
            loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy
_with_logits(
                logits=logits_train, labels=_y))
            optimizer = tf.train.AdamOptimizer(learning_rate=lea
rning_rate)
            grads = optimizer.compute_gradients(loss_op)

            # Only first GPU compute accuracy
            if i == 0:
                # Evaluate model (with test logits, for dropout
to be disabled)
                correct_pred = tf.equal(tf.argmax(logits_test, 1
), tf.argmax(_y, 1))
                accuracy = tf.reduce_mean(tf.cast(correct_pred,
tf.float32))

            reuse_vars = True
            tower_grads.append(grads)
```

```
    tower_grads = average_gradients(tower_grads)
    train_op = optimizer.apply_gradients(tower_grads)

    # Initializing the variables
    init = tf.global_variables_initializer()

    # Launch the graph
    with tf.Session() as sess:
        sess.run(init)
        step = 1
        # Keep training until reach max iterations
        for step in range(1, num_steps + 1):
            # Get a batch for each GPU
            batch_x, batch_y = mnist.train.next_batch(batch_size
 * num_gpus)
            # Run optimization op (backprop)
            ts = time.time()
            sess.run(train_op, feed_dict={X: batch_x, Y: batch_y
})
            te = time.time() - ts
            if step % display_step == 0 or step == 1:
                # Calculate batch loss and accuracy
                loss, acc = sess.run([loss_op, accuracy], feed_d
ict={X: batch_x,

     Y: batch_y})
                print("Step " + str(step) + ": Minibatch Loss= "
 + \
                      "{:.4f}".format(loss) + ", Training Accura
cy= " + \
                      "{:.3f}".format(acc) + ", %i Examples/sec"
 % int(len(batch_x)/te))
            step += 1
        print("Optimization Finished!")

        # Calculate accuracy for 1000 mnist test images
        print("Testing Accuracy:", \
            np.mean([sess.run(accuracy, feed_dict={X: mnist.test
.images[i:i+batch_size],
            Y: mnist.test.labels[i:i+batch_size]}) for i in rang
e(0, len(mnist.test.images), batch_size)]))
```

```
Step 1: Minibatch Loss= 2.4077, Training Accuracy= 0.123, 682 Ex
amples/sec
Step 10: Minibatch Loss= 1.0067, Training Accuracy= 0.765, 6528
Examples/sec
Step 20: Minibatch Loss= 0.2442, Training Accuracy= 0.945, 6803
Examples/sec
Step 30: Minibatch Loss= 0.2013, Training Accuracy= 0.951, 6741
Examples/sec
Step 40: Minibatch Loss= 0.1445, Training Accuracy= 0.962, 6700
Examples/sec
Step 50: Minibatch Loss= 0.0940, Training Accuracy= 0.971, 6746
Examples/sec
Step 60: Minibatch Loss= 0.0792, Training Accuracy= 0.977, 6627
Examples/sec
Step 70: Minibatch Loss= 0.0593, Training Accuracy= 0.979, 6749
Examples/sec
Step 80: Minibatch Loss= 0.0799, Training Accuracy= 0.984, 6368
Examples/sec
Step 90: Minibatch Loss= 0.0614, Training Accuracy= 0.988, 6762
Examples/sec
Step 100: Minibatch Loss= 0.0716, Training Accuracy= 0.983, 6338
 Examples/sec
Step 110: Minibatch Loss= 0.0531, Training Accuracy= 0.986, 6504
 Examples/sec
Step 120: Minibatch Loss= 0.0425, Training Accuracy= 0.990, 6721
 Examples/sec
Step 130: Minibatch Loss= 0.0473, Training Accuracy= 0.986, 6735
 Examples/sec
Step 140: Minibatch Loss= 0.0345, Training Accuracy= 0.991, 6636
 Examples/sec
Step 150: Minibatch Loss= 0.0419, Training Accuracy= 0.993, 6777
 Examples/sec
Step 160: Minibatch Loss= 0.0602, Training Accuracy= 0.984, 6392
 Examples/sec
Step 170: Minibatch Loss= 0.0425, Training Accuracy= 0.990, 6855
 Examples/sec
Step 180: Minibatch Loss= 0.0107, Training Accuracy= 0.998, 6804
 Examples/sec
Step 190: Minibatch Loss= 0.0204, Training Accuracy= 0.995, 6645
 Examples/sec
Step 200: Minibatch Loss= 0.0296, Training Accuracy= 0.993, 6747
 Examples/sec
Optimization Finished!
Testing Accuracy: 0.990671
```